

# Lecture 8

# Code Refactoring

# What is Code Refactoring

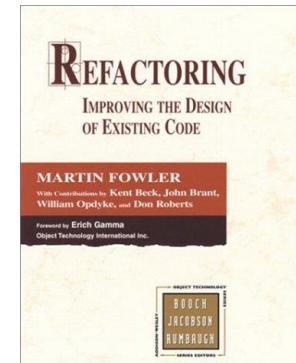
- "... Process of changing a software system in such a way that it **does not alter the external behaviour** of the code yet improves its internal structure." – Fowler.
- Simply put, it is just **cleaning up** your code

# Refactoring

- Basic metaphor:
  - Start with an existing code base and make it better.
  - Change the internal structure while preserving the overall semantics
    - *i.e.*, rearrange the “factors” but end up with the same final “product”
- The idea is that you should **improve the code** in some significant way. For example:
  - Reducing near-duplicate code
  - Improved cohesion, lessened coupling
  - Improved parameterization, understandability, maintainability, flexibility, abstraction, efficiency, *etc ...*

# Refactoring

- Reference:
  - *Refactoring: Improving the Design of Existing Code*,  
by Martin Fowler (*et al.*), 1999, Addison-Wesley
- Fowler, Beck, *et al.* are big wheels in the OOA&D crowds
  - Smalltalk experience
  - OO design patterns
  - XP (extreme programming)
- Book is very good
  - Ideas have been around for a long time tho.
    - Experienced OO programmers will have a pretty good handle on most of the ideas
  - Mostly it's a catalogue of transformations that you can perform on your code, with motivation, explanation, variations, and examples e.g., *Extract Interface*, *Move Method*, *Extract Class*



# Refactoring

- Book has a catalogue of:
  - 22 “bad smells in code”  
*i.e.*, things to look out for, “anti-patterns”
    - ***This font and colour indicates the name of a bad smell.***
  - 72 “refactorings”  
*i.e.*, what to do when you find them
    - ***This font and colour indicates the name of a refactoring.***
  - We will look at some of the bad smells and what to do about them.

# Some advice from Fowler

- *"When should I refactor? How often?  
How much time should I dedicate to it?"*
  
- It's not something you should dedicate two weeks for every six months ...
- ... rather, you should do it as you develop!
  - Refactor when you recognize a warning sign (a "bad smell") and know what to do
  - ... when you **add a function**
    - Likely it's not an island unto itself
  - ... when you **fix a bug**
    - Is the bug symptomatic of a design flaw?
  - ... when you do **a code review**
    - A good excuse to re-evaluate your designs, share opinions.

# XP and Refactoring

- **Simple design** – system is designed as simply as possible (extra complexity removed as soon as found)
- **Refactoring** – programmers continuously restructure the system without changing its behavior to remove duplication and simplify
  - Any programming construct can be made more abstract ... but that's not necessarily a good thing.
    - Generality (flexibility) costs too
  - Don't spin your wheels designing and coding the most abstract system you can imagine.
    - Practise Just-in-Time abstraction.
    - *Expect* that you will be re-arranging your code constantly. Don't worry about it. Embrace it.

# Bad smells in code

- Duplicated Code
- Long Method
- Large Class
- Long Parameter List
- Divergent Change
- Shotgun Surgery
- Feature Envy
- Data Clumps
- Primitive Obsession
- Switch Statements
- Parallel Inheritance Hierarchies
- Lazy Class
- Speculative Generality
- Temporary Field
- Message Chains
- Middle Man
- Inappropriate Intimacy
- Alternative Classes with Different Interfaces
- Incomplete Library Class
- Data Class
- Refused Bequest
- Comments

# Bad smells in code

## Duplicated code

- “The #1 bad smell”
- Same expression in two methods in the same class
  - Make it a private ancillary routine and parameterize it  
*(Extract method)*

```
int array1[]; int array2[];  
int sum1 = 0; int sum2 = 0;  
int average1 = 0; int average2 = 0;
```

```
Void A(){
```

```
...
```

```
    for (int i = 0; i < 4; i++) {  
        sum1 += array1[i]; }  
    average1 = sum1/4;
```

```
...  
}
```

```
Void B() {
```

```
...
```

```
    for (int i = 0; i < 4; i++) {  
        sum2 += array2[i]; }  
    average2 = sum2/4;
```

```
...
```

```
int array1[]; int array2[];  
Int average1 = 0; int average2 = 0;  
  
void A(){  
...  
    average1 = calcAverage (array1);  
...  
}
```

```
void B() {  
...  
    average2 = calcAverage (array2);  
...  
}
```

```
int calcAverage (int* Array_of_4) {  
    int sum = 0;  
    for (int i = 0; i < 4; i++)  
    {  
        sum += Array_of_4[i];  
    }  
    return sum/4;  
}
```

# Bad smells in code

## ■ **Duplicated code**

- Same code in two *related* classes.
  - Push commonalities into closest mutual ancestor and parameterize  
*(Form template method)*
- Same code in two *unrelated* classes.
  - Ought they be related?
    - Introduce abstract parent *(Extract class, Pull up method)*
  - Does the code really belongs to just one class?
    - Make the other class into a client *(Extract method)*

# Bad smells in code

## ■ *Long method*

- Often a sign of:
  - Trying to do too many things
  - Poorly thought out abstractions and boundaries
- Best to think carefully about the major tasks and how they inter-relate.
  - Break up into smaller private methods within the class  
*(Extract method)*
  - Delegate subtasks to subobjects that “know best” (*i.e.*, template method DP)  
*(Extract class/method, Replace data value with object)*

# Bad smells in code

## ■ ***Long method***

- Fowler's heuristic:
  - *When you see a comment, make a method.*
  - Often, a comment indicates:
    - The next major step
    - Something non-obvious whose details detract from the clarity of the routine as a whole.
  - In either case, this is a good spot to "break it up".

Take a look at sample long method code in pdf

# Bad smells in code

- *Large class*



# Bad smells in code

- ***Large class***
  - Too many different subparts and methods
    - *Which means lack of ???*
  - Two step solution:
    1. Gather up the little pieces into aggregate subparts.  
**(Extract class, replace data value with object)**
    2. Delegate methods to the new subparts.  
**(Extract method)**
  - Likely, you'll notice some unnecessary subparts that have been hiding in the forest!

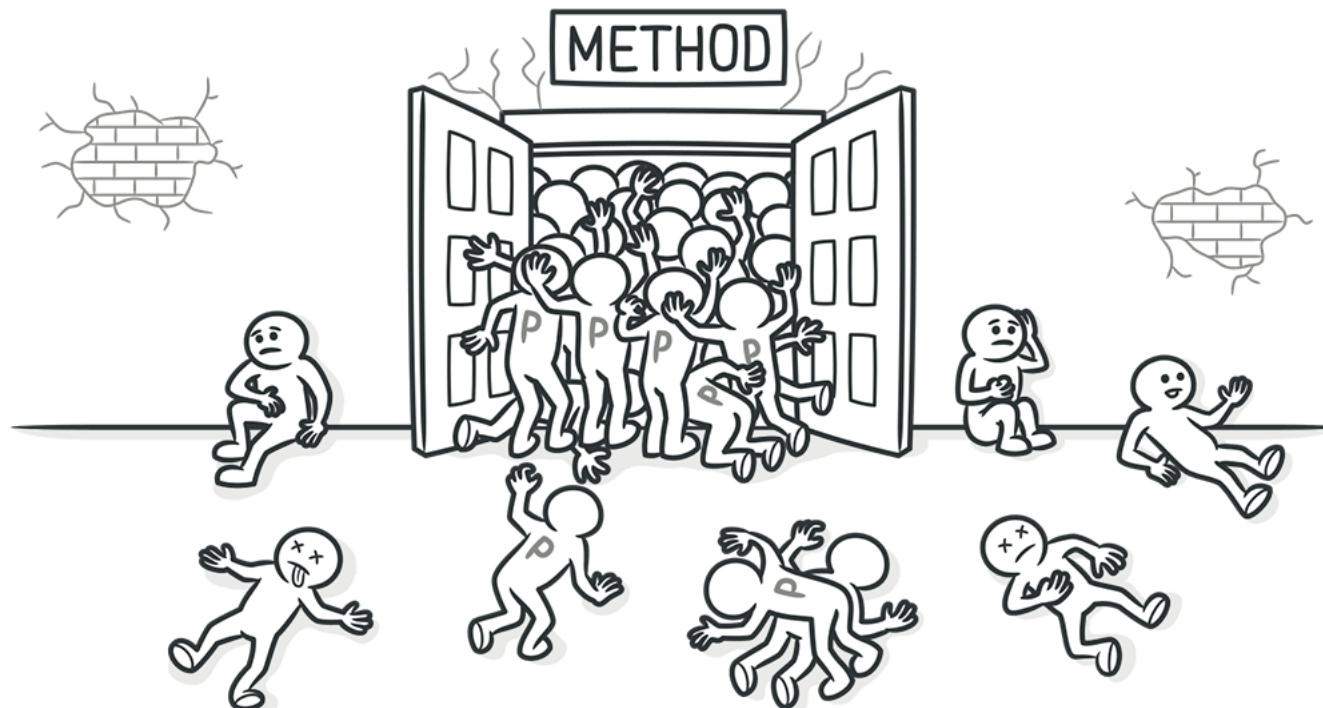
# Bad smells in code

- ***Large class***
  - Counter example:
    - **Library classes** often have large, fat interfaces (many methods, many parameters, lots of overloading)
      - If the many methods exist *for the purpose of flexibility*, that's OK in a library class.

# Bad smells in code

- ***Long parameter list***

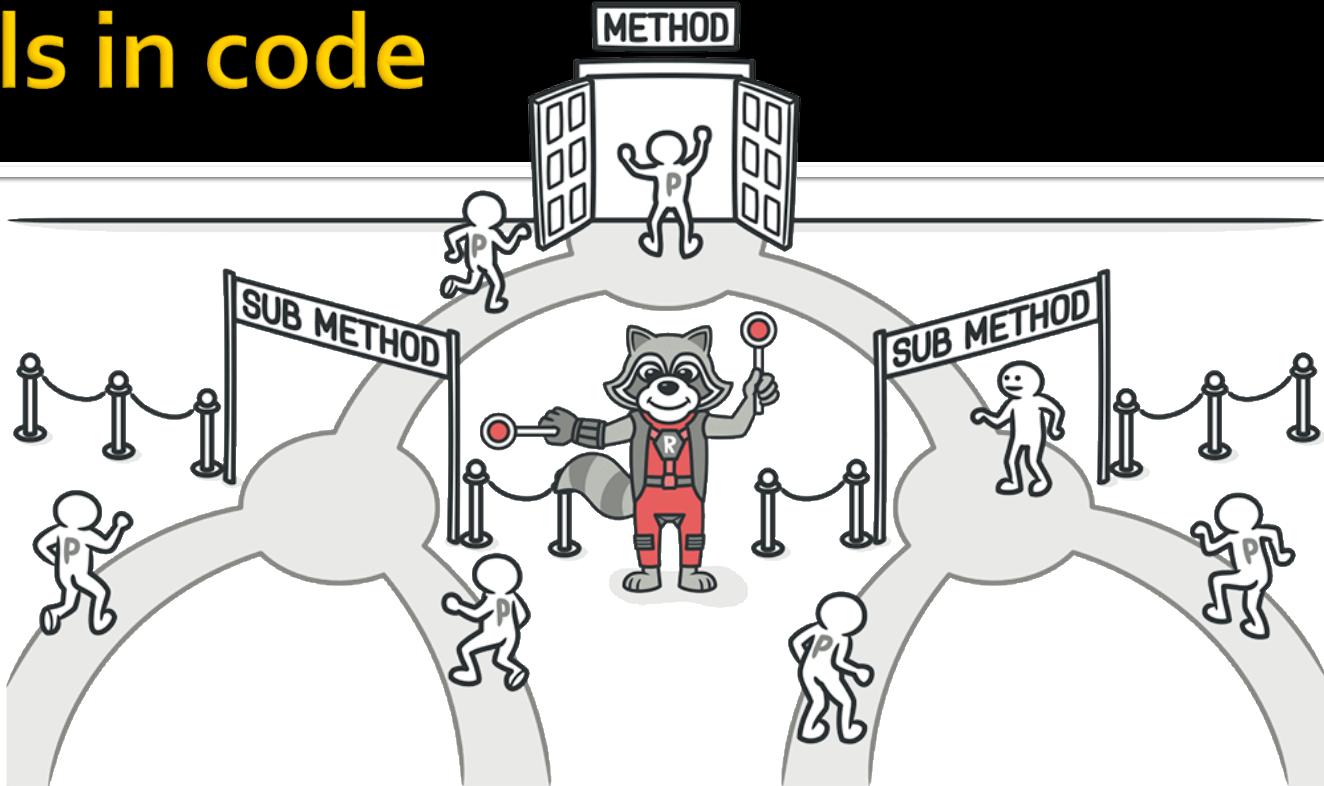
- More than three or four parameters for a method



# Bad smells in code

- ***Long parameter list***
  - Long parameter lists make methods difficult for clients to understand
  - This is often a symptom of
    - Trying to do too much
    - with too many disparate subparts
  - In the old days, structured programming taught the use of parameterization as a cure for global variables.
    - With OOP, objects have mini-islands of state that can be reasonably treated as “global” to the methods (yet are still hidden from the rest of the program).

# Bad smells in code



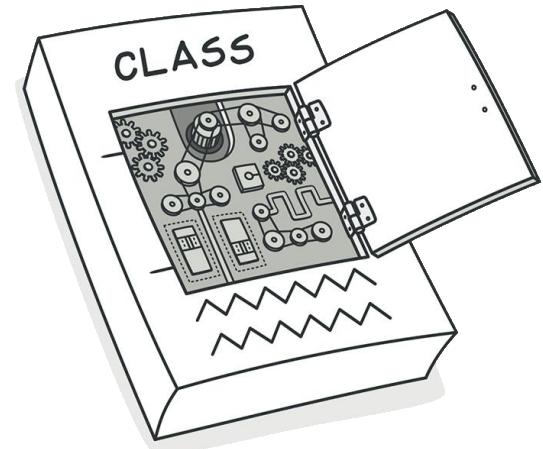
## Solutions to Long Parameter List

- Trying to do too much?
  - Break up into sub-tasks  
*(Extract method)*
- ... with too many disparate subparts?
  - Gather up parameters into aggregate subparts
  - Your method interfaces will be much easier to understand!  
*(Preserve whole object, introduce parameter object)*

# Bad smells in code

## Divergent change

- Occurs when one class is commonly changed in different ways for different reasons
- Likely, this class is trying to do too much and contains too many unrelated subparts
  - *Similar to the code smell ???*
- Over time, some classes develop a “God Complex”
  - They acquires details/ownership of subparts that rightly belong elsewhere
- This is a sign of *poor cohesion*
  - Unrelated elements in the same container
- Solution:
  - Break it up, reshuffle, reconsider relationships and responsibilities (**Extract class**)

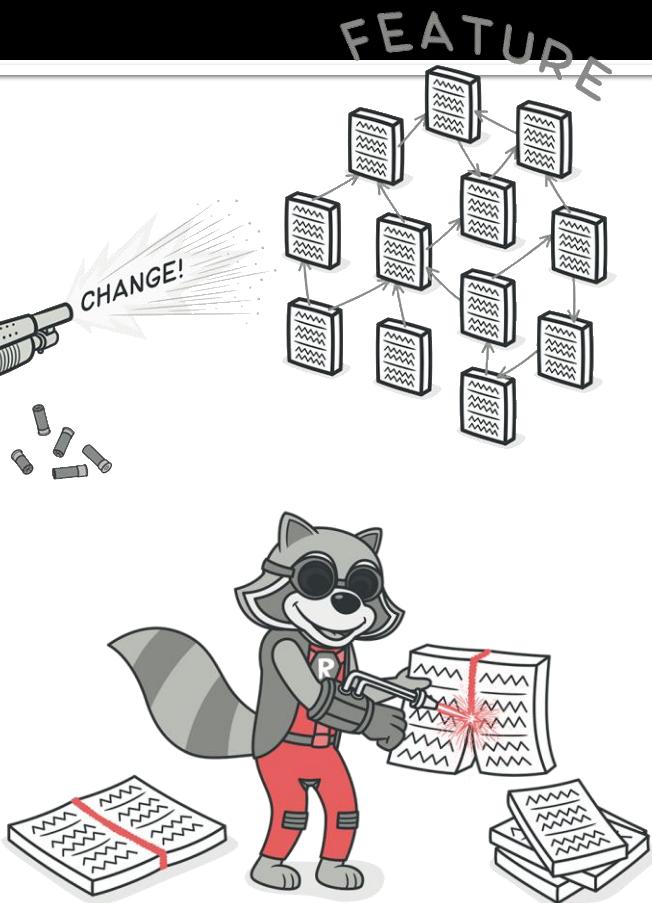
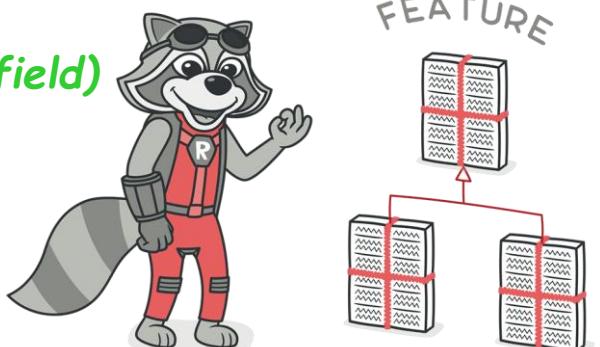


# Bad smells in code

## Shotgun Surgery

- Each time you want to make a single, seemingly coherent change, you have to change lots of classes in little ways
- Also a classic sign of poor cohesion
  - Related elements are *not* in the same container!
- Solution:
  - Look to do some gathering, either in a new or existing class.

(Move method/field)



# Bad smells in code

- ***Feature envy***
  - A method seems more interested in another class than the one it's defined in
    - e.g., a method `A::m()` calls lots of get/set methods of class `B`
  - Solution:
    - Move `m()` (or part of it) into `B!`  
*(Move method/field, extract method)*

# Bad smells in code

## ■ *Data Clumps*

- You see a set of variables that seem to “hang out” together e.g., passed as parameters, changed/accessed at the same time
- Usually, this means that there’s a coherent subobject just waiting to be recognized and encapsulated

```
void Scene::setTitle (string titleText,  
                     int titleX, int titleY,  
                     Colour titleColour) {...}  
  
void Scene::getTitle (string& titleText,  
                     int& titleX, int& titleY,  
                     Colour& titleColour) {...}
```

# Bad smells in code

- **Data Clumps**
  - In the example, a `Title` class is dying to be born
  - If a client knows how to change a title's `x`, `y`, `text`, and `colour`, then it knows enough to be able to "roll its own" `Title` objects.
    - However, this does mean that the client now has to talk to another class.
  - This will greatly shorten and simplify your parameter lists (which aids understanding) and makes your class conceptually simpler too.

*(Preserve whole object, extract class, introduce parameter object)*

# Bad smells in code

## ■ ***Primitive Obsession***

- All subparts of an object are instances of primitive types (int, string, bool, double, etc.)  
e.g., dates, currency, SIN, tel.#, ISBN, special string values
- Often, these small objects have interesting and non-trivial constraints that can be modelled  
e.g., fixed number of digits/chars, check digits, special values
- Solution:
  - Create some “small classes” that can validate and enforce the constraints.  
*(Replace data value with object, extract class, introduce parameter object)*

# Bad smells in code

## ■ *Message chains*

- Client asks an object which asks a subobject, which asks a subobject, ...
  - Multi-layer “drill down” may result in sub-sub-sub-objects being passed back to requesting client.
  - Sounds like the client already has an understanding of the structure of the object, even if it is going through appropriate intermediaries.
- Probably need to rethink abstraction ...
  - Why is a deeply nested subpart surfacing?
  - Why is the subpart so simple that it’s useful far from home?  
*(Hide delegate)*

# Bad smells in code

- **Data class**
  - Class consists of (simple) data fields and simple get/set methods only.
  - Solution:
    - Have a look at usage patterns in the clients
    - Try to abstract some commonalities of usage into methods of the data class and move some functionality over
  - Data classes can be quite reasonable, if used judiciously.
    - In C++, often use structs to model data classes.
  - "*Data classes are like children. They are OK as a starting point, but to participate as a grownup object, they need to take on some responsibility.*"  
*(Extract/move method)*

# Bad smells in code

## ■ **Comments**

- In the context of refactoring, Fowler claims that long comments are often a sign of opaque, complicated, inscrutable code.
  - They aren't against comments so much as in favour of self-evident coding practices.
  - Rather than explaining opaque code, restructure it!

*(Extract method/class, [many others applicable] ...)*

- Comments are best used to document rationale
  - i.e., explain *why* you picked one approach over another.

# Summary

- Fowler *et al.*'s *Refactoring* is a well-written book that summarizes a lot of “best practices” of OOD/OOP with nice examples.
  - Many books on OOD heuristics are vague and lack concrete advice.
  - Most of the advice in this book is aimed at low-level OO programming.
    - *i.e.*, loops, variables, method calls, and class definitions.
  - Next obvious step up in abstraction/scale is to OODPs.
    - *i.e.*, collaborating classes
- This is an excellent book for the intermediate-level OO programmer.
  - Experienced OO programmers will have discovered a lot of the techniques already on their own.

# Small Refactoring Techniques

- Composing Methods
  - Extract Method
  - Inline Method
  - Inline Temp
  - Replace Temp with Query
  - Introduce Explaining Variable
  - Split Temporary Variable
  - Remove Assignments to Parameters
  - Replace Method with Method Object
  - Substitute Algorithm
- Moving Features Between Objects
  - Move Method
  - Move Field
  - Extract Class
  - Inline Class
  - Hide Delegate
  - Remove Middle Man
  - Introduce Foreign Method
  - Introduce Local Extension.....

# Small Refactoring Techniques

- Organizing Data
  - Self Encapsulate Field
  - Replace Data Value with Object
  - Change Value to Reference
  - Change Reference to Value
  - Replace Array with Object
  - Duplicate Observed Data
  - Change Unidirectional Association to Bidirectional
  - Change Bidirectional Association to Unidirectional
  - Replace Magic Number with Symbolic Constant
  - Encapsulate Field
  - Encapsulate Collection
  - Replace Record with Data Class
  - Replace Type Code with Class
  - Replace Type Code with Subclasses
  - Replace Type Code with State/Strategy
  - Replace Subclass with Fields
- Simplifying Conditional Expressions
  - Decompose Conditional
  - Consolidate Conditional Expression
  - Consolidate Duplicate Conditional Fragments
  - Remove Control Flag
  - Replace Nested Conditional with Guard Clauses
  - Replace Conditional with Polymorphism
  - Introduce Null Object
  - Introduce Assertion

# Small Refactoring Techniques

- Making Method Calls Simpler
  - Rename Method
  - Add Parameter
  - Remove Parameter
  - Separate Query from Modifier
  - Parameterize Method
  - Replace Parameter with Explicit Methods
  - Preserve Whole Object
  - Replace Parameter with Method
  - Introduce Parameter Object
  - Remove Setting Method
  - Hide Method
  - Replace Constructor with Factory Method
  - Encapsulate Downcast
  - Replace Error Code with Exception
  - Replace Exception with Test
- Dealing with Generalization
  - Pull Up Field
  - Pull Up Method
  - Pull Up Constructor Body
  - Push Down Method
  - Push Down Field
  - Extract Subclass
  - Extract Superclass
  - Extract Interface
  - Collapse Hierarchy
  - Form Template Method
  - Replace Inheritance with Delegation
  - Replace Delegation with Inheritance

# Extract Method

```
void printOwing(double amount) {  
    printBanner();  
  
    //print details  
    System.out.println ("name:" + _name);  
    System.out.println ("amount" + amount);  
}  
  
void printOwing(double amount) {  
    printBanner();  
  
    printDetails(amount);  
}  
  
void printDetails (double amount) {  
    System.out.println ("name:" + _name);  
    System.out.println ("amount" + amount);  
}
```

*You have a code fragment that can be grouped together.*

**Turn the fragment into a method whose name explains the purpose of the method.**

# Extract Method

```
void printOwing() {  
    Enumeration e = _orders.elements();  
    double outstanding = 0.0;  
  
    // print banner  
    System.out.println ("*****");  
    System.out.println ("***** Customer Owes *****");  
    System.out.println ("*****");  
  
    // calculate outstanding  
    while (e.hasMoreElements()) {  
        Order each = (Order) e.nextElement();  
        outstanding += each.getAmount();  
    }  
  
    //print details  
    System.out.println ("name:" + _name);  
    System.out.println ("amount" + outstanding);  
}
```

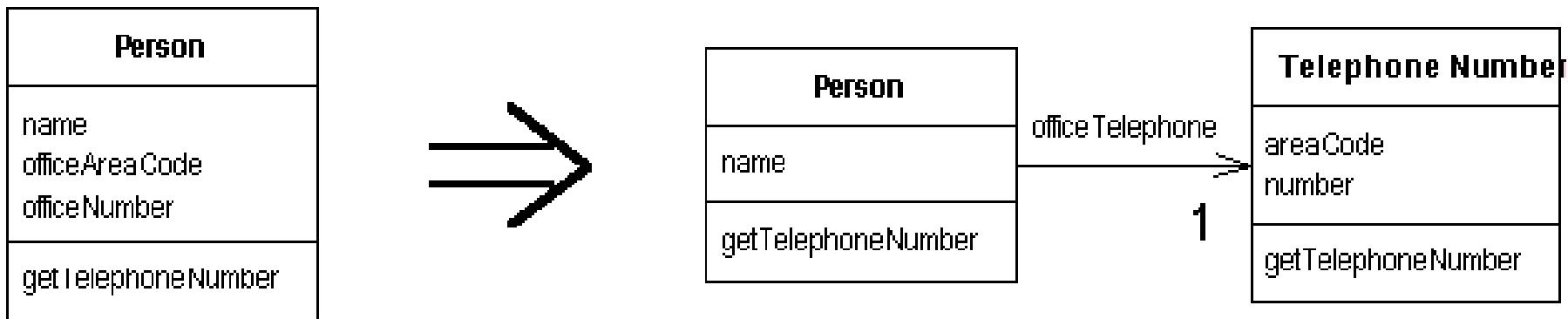


```
void printOwing() {  
    printBanner();  
    double outstanding = getOutstanding();  
    printDetails(outstanding);  
}  
  
void printBanner() {  
    // print banner  
    System.out.println ("*****");  
    System.out.println ("***** Customer Owes *****");  
    System.out.println ("*****");  
}  
  
double getOutstanding() {  
    Enumeration e = _orders.elements();  
    double outstanding = 0.0;  
    while (e.hasMoreElements()) {  
        Order each = (Order) e.nextElement();  
        outstanding += each.getAmount();  
    }  
    return outstanding;  
}  
  
void printDetails (double outstanding) {  
    System.out.println ("name:" + _name);  
    System.out.println ("amount" + outstanding);  
}
```

# Extract class

You have one class doing work that should be done by two.

Create a new class and move the relevant fields and methods from the old class into the new class.



# Extract class + Move Field

```
class Person...
public String getName() {
    return _name;
}

public String getTelephoneNumber() {
    return "(" + _officeAreaCode + ")" + _officeNumber;
}

String getOfficeAreaCode() {
    return _officeAreaCode;
}

void setOfficeAreaCode(String arg) {
    _officeAreaCode = arg;
}

String getOfficeNumber() {
    return _officeNumber;
}

void setOfficeNumber(String arg) {
    _officeNumber = arg;
}

private String _name;
private String _officeAreaCode;
private String _officeNumber;
```

```
class TelephoneNumber {
    String getAreaCode() {
        return _areaCode;
    }

    void setAreaCode(String arg) {
        _areaCode = arg;
    }

    private String _areaCode;
}
```

```
class Person...
private TelephoneNumber _officeTelephone =
new TelephoneNumber();

public String getTelephoneNumber() {
    return "(" + getOfficeAreaCode() + ")" +
        _officeNumber;
}

String getOfficeAreaCode() {
    return _officeTelephone.getAreaCode();
}

void setOfficeAreaCode(String arg) {
    _officeTelephone.setAreaCode(arg);
}
```

# Extract Class + Move Method

```
class TelephoneNumber {  
    String getAreaCode() {  
        return _areaCode;  
    }  
    void setAreaCode(String arg) {  
        _areaCode = arg;  
    }  
    private String _areaCode;  
}
```

```
class Person...  
private TelephoneNumber _officeTelephone =  
new TelephoneNumber();  
  
public String getTelephoneNumber() {  
    return "(" + getOfficeAreaCode() + ") " +  
    _officeNumber;  
}  
String getOfficeAreaCode() {  
    return _officeTelephone.getAreaCode();  
}  
void setOfficeAreaCode(String arg) {  
    _officeTelephone.setAreaCode(arg);  
}
```

```
class TelephoneNumber...  
public String getTelephoneNumber() {  
    return "(" + _areaCode + ") " + _number;  
}  
String getAreaCode() {  
    return _areaCode;  
}  
void setAreaCode(String arg) {  
    _areaCode = arg;  
}  
String getNumber() {  
    return _number;  
}  
void setNumber(String arg) {  
    _number = arg;  
}  
private String _number;  
private String _areaCode;
```

```
class Person...  
public String getName() {  
    return _name;  
}  
public String getTelephoneNumber(){  
    return _officeTelephone.getTelephoneNumber();  
}  
private String _name;  
private TelephoneNumber _officeTelephone =  
new TelephoneNumber();
```

# Decompose Conditional

You have a complicated conditional (if-then-else) statement.

*Extract methods from the condition, then part, and else parts.*

```
if (date.before(SUMMER_START) || date.after(SUMMER_END))  
    charge = quantity * _winterRate + _winterServiceCharge;  
else charge = quantity * _summerRate;
```



```
if (notSummer(date))  
    charge = winterCharge(quantity);  
else charge = summerCharge(quantity);
```

# Consolidate Conditional Expression

You have a sequence of conditional tests with the same result.

*Combine them into a single conditional expression and extract it.*

```
double disabilityAmount() {  
    if (_seniority < 2) return 0;  
    if (_monthsDisabled > 12) return 0;  
    if (_isPartTime) return 0;  
    // compute the disability amount
```



```
double disabilityAmount() {  
    if (isNotEligibleForDisability()) return 0;  
    // compute the disability amount
```

# Replace Array with Object

You have an array in which certain elements mean different things.

*Replace the array with an object that has a field for each element.*

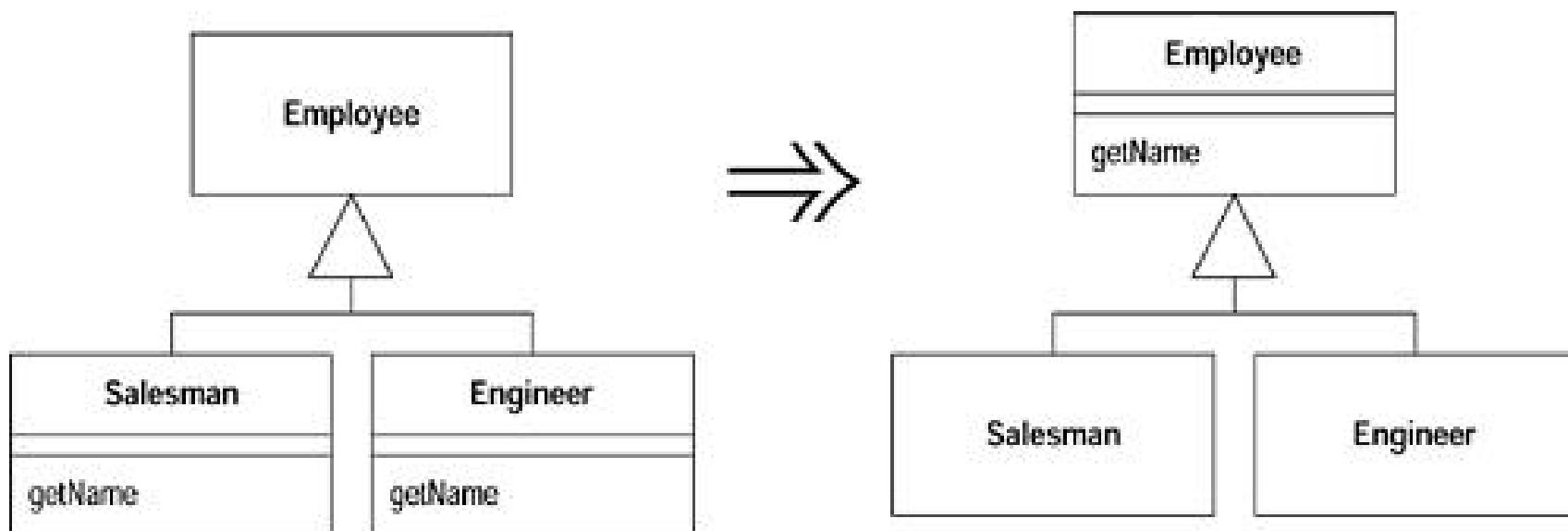
```
String[] row = new String[3];  
row [0] = "Liverpool";  
row [1] = "15";
```



```
Performance row = new Performance();  
row.setName("Liverpool");  
row.setWins("15");
```

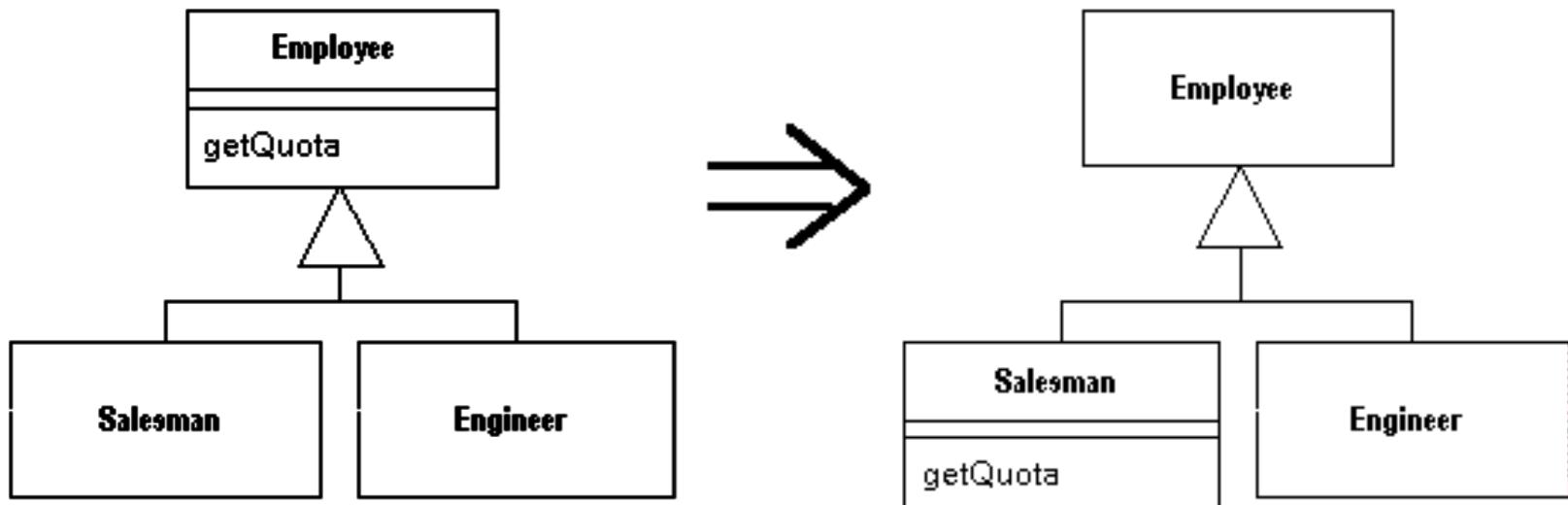
# Pull Up Method

*You have methods with identical results on subclasses.*  
**Move them to the superclass.**



# Push Down Method

*Behavior on a superclass is relevant only for some of its subclasses.*  
**Move it to those subclasses.**

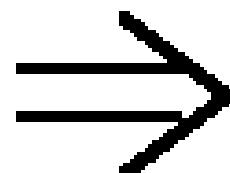


# Introduce Parameter Object

You have a group of parameters that naturally go together.

*Replace them with an object.*

Customer
amountInvoicedIn(start: Date, end: Date)
amountReceivedIn(start: Date, end: Date)
amountOverdueIn(start: Date, end: Date)



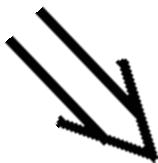
Customer
amountInvoicedIn(Date Range)
amountReceivedIn(Date Range)
amountOverdueIn(Date Range)

# Preserve Whole Object

You are getting several values from an object and passing these values as parameters  
in a method call

*Send the whole object instead*

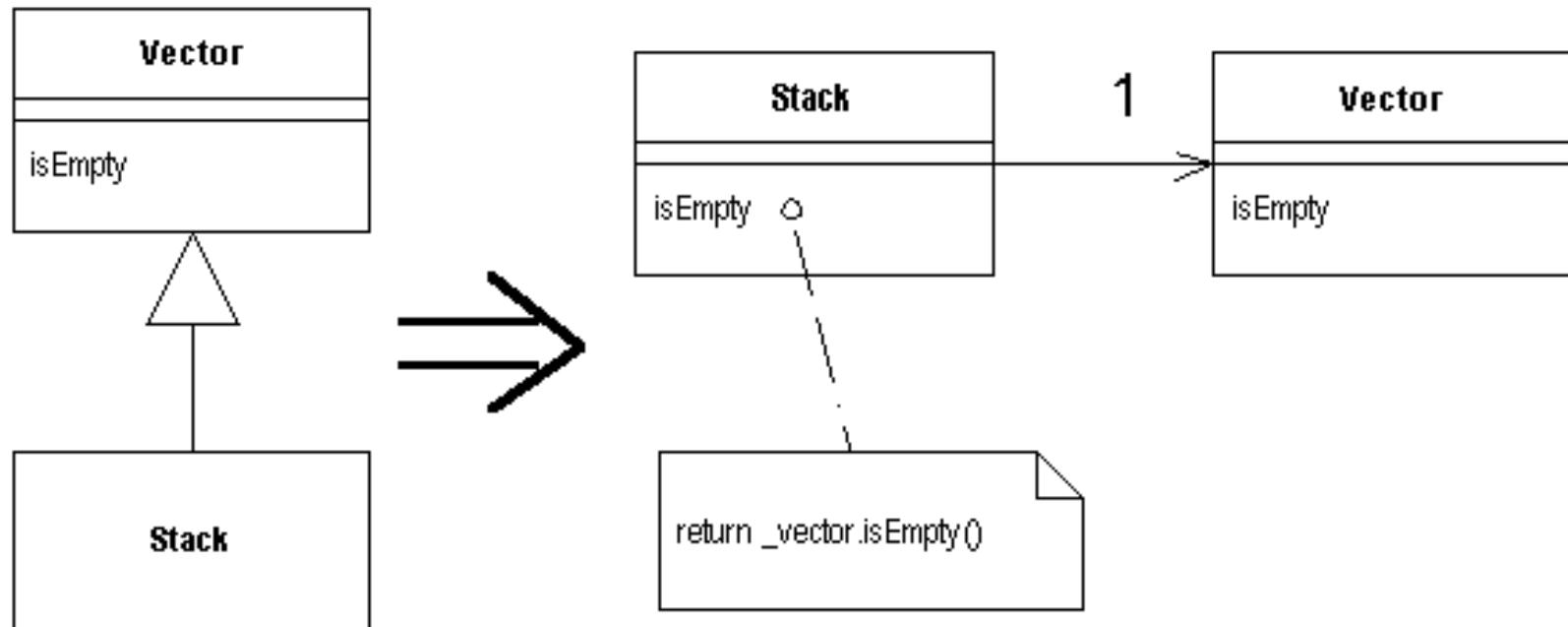
```
int low = daysTempRange.getLow();
int high = daysTempRange.getHigh();
withinPlan = plan.withinRange(low, high);
```



```
withinPlan = plan.withinRange(daysTempRange);
```

# Replace Inheritance with Delegation

*subclass uses only part of a superclasses interface or does not want to inherit data.*  
**Create a field for the superclass, adjust methods to delegate to the superclass,**  
**and remove the subclassing.**



# Big Refactoring

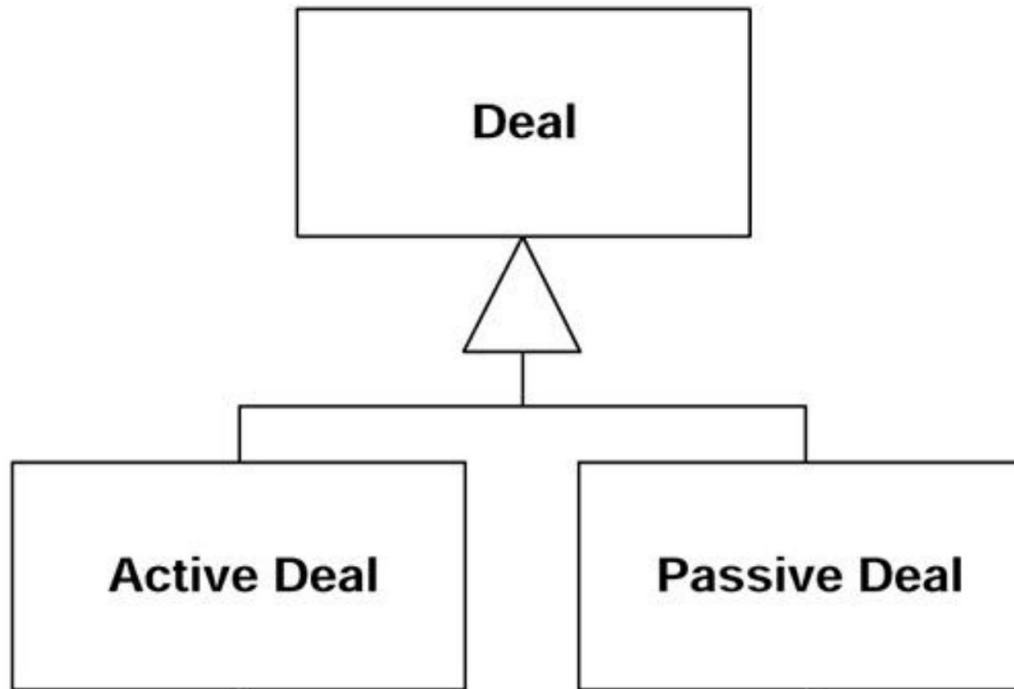
- Tease apart inheritance
- Extract hierarchy
- Convert procedural design to objects
- Separate domain from presentation

# Big Refactoring

## *Tease apart inheritance*

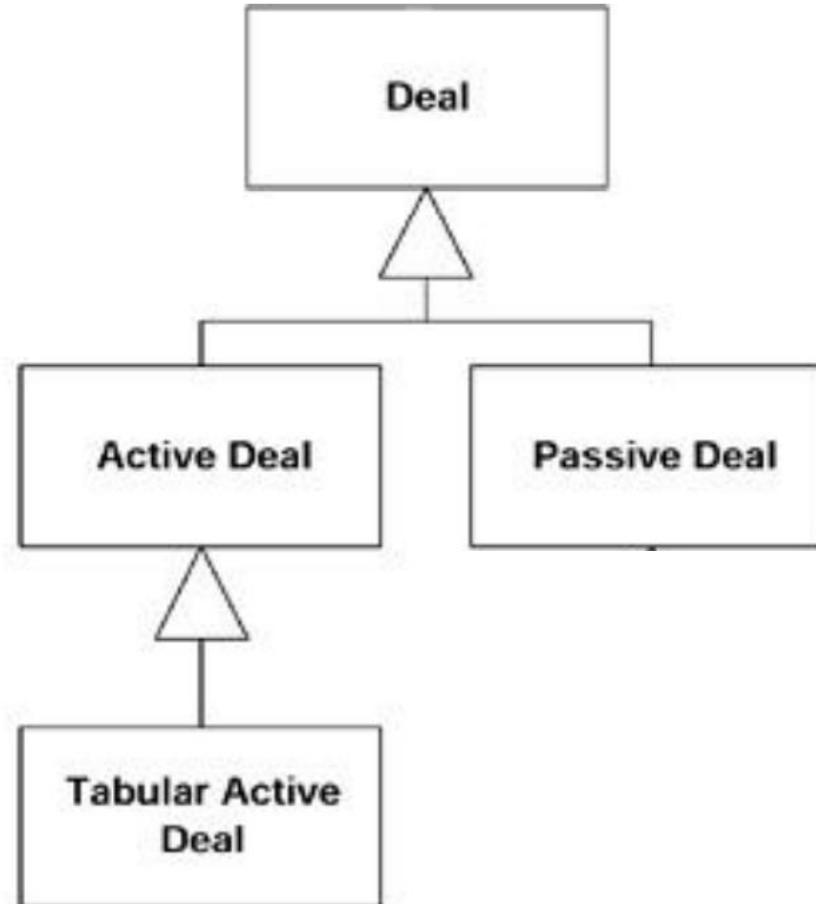
You have an inheritance hierarchy that is doing two jobs at once.

*Create two hierarchies and use delegation to invoke one from the other.*



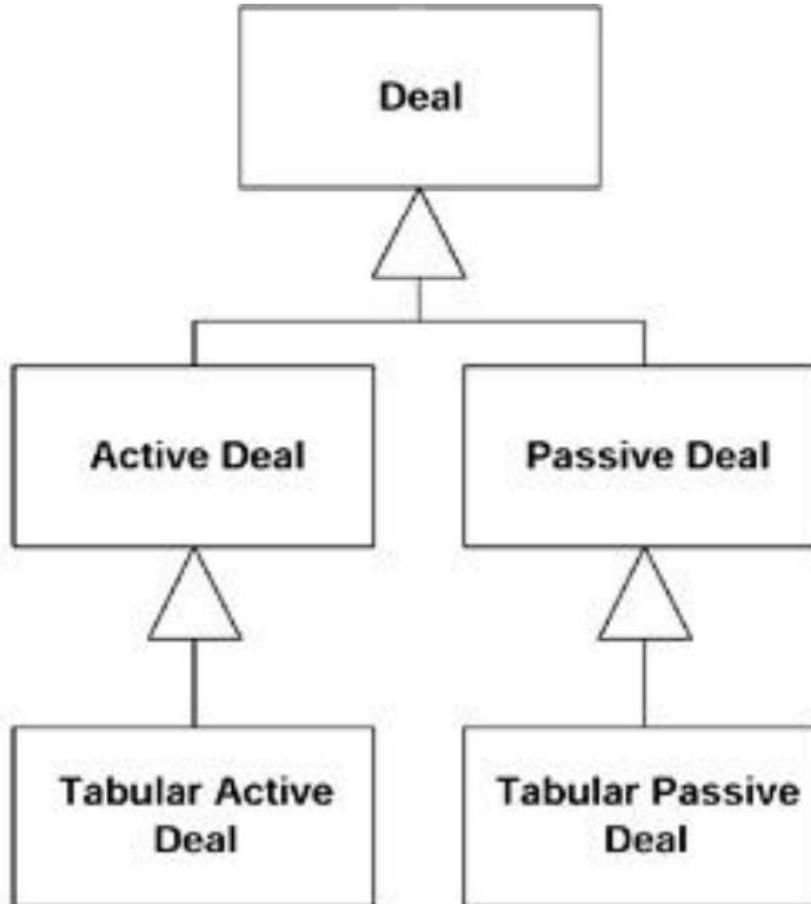
# Big Refactoring

## *Tease apart inheritance*



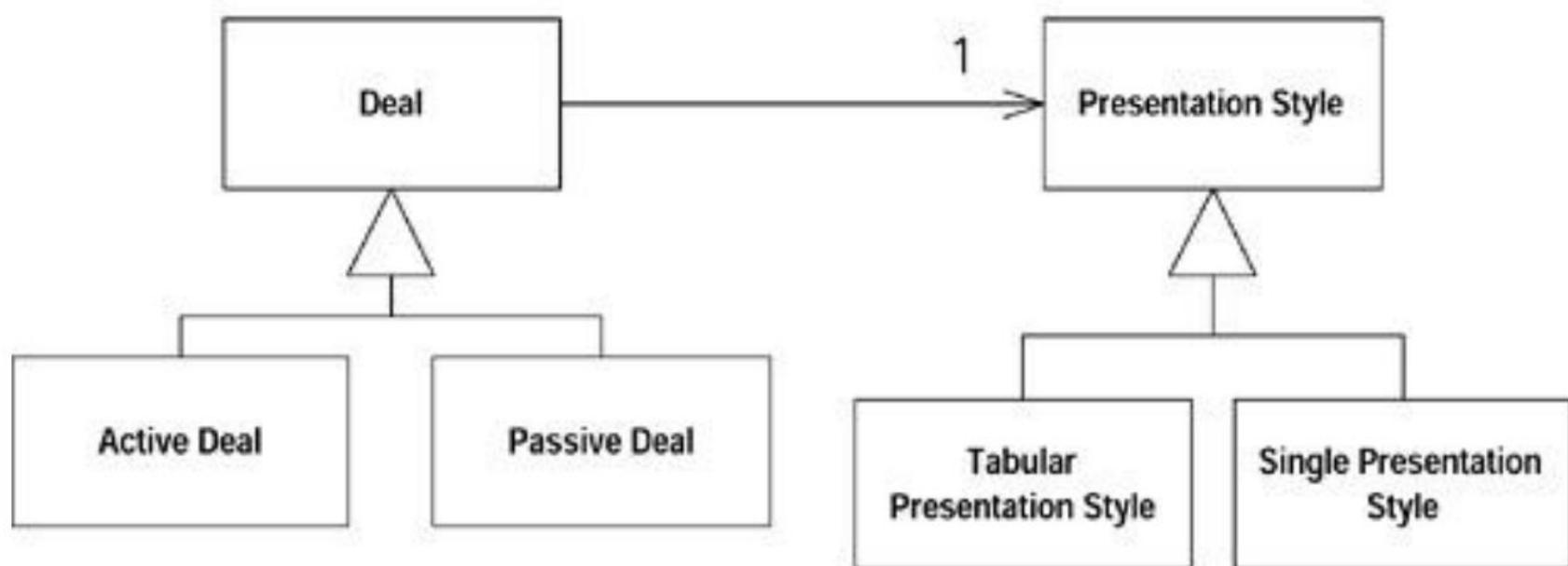
# Big Refactoring

## *Tease apart inheritance*



# Big Refactoring

## *Tease apart inheritance*



# Big Refactoring

## *Extract Hierarchy*

- You have a class that is doing too much work, at least in part through many conditional statements.

*Create a hierarchy of classes in which each subclass represents a special case*

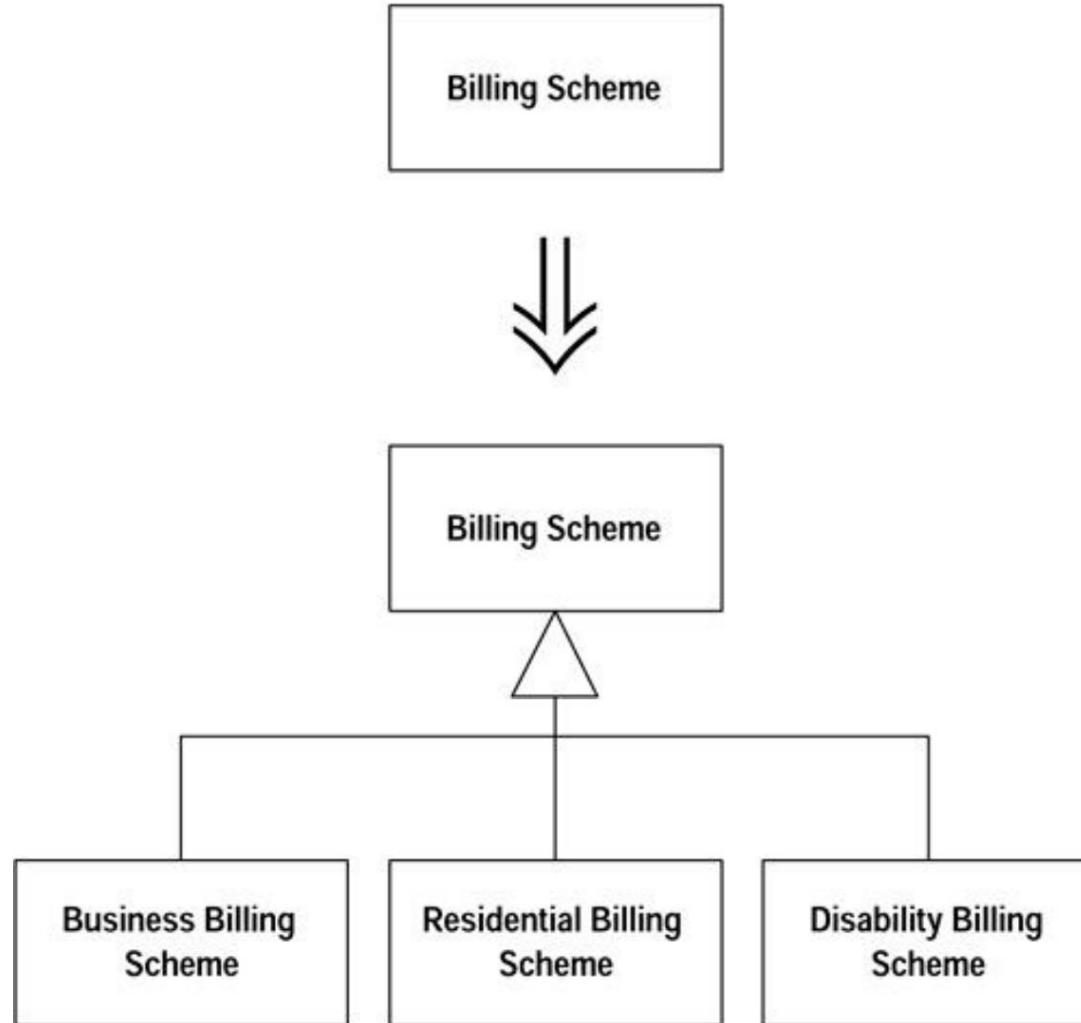
# Big Refactoring

## *Extract Hierarchy*

Billing Scheme

# Big Refactoring

## *Extract Hierarchy*



# Big Refactoring

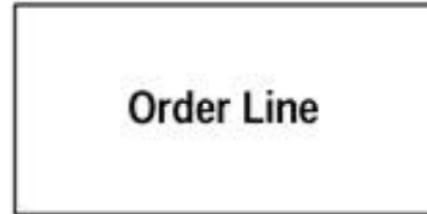
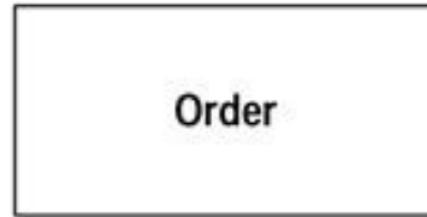
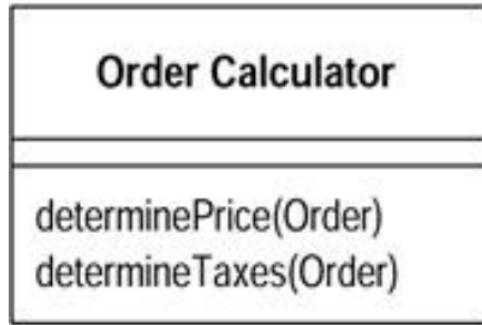
## *Convert Procedural Design to Objects*

- You have code written in a procedural style.

*Turn the data records into objects, break up the behavior, and move the behavior to the objects.*

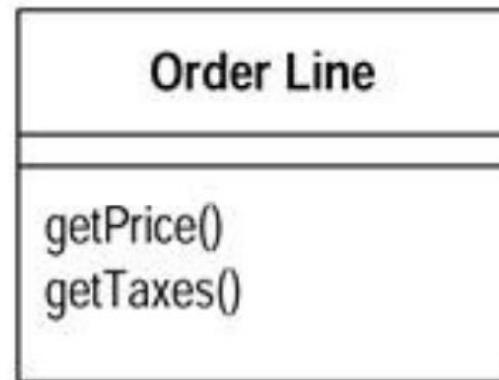
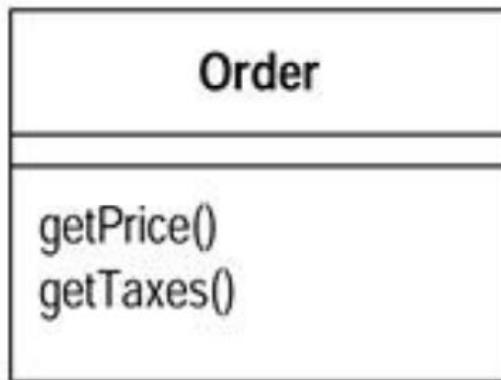
# Big Refactoring

*Convert Procedural Design to Objects*



# Big Refactoring

*Convert Procedural Design to Objects*



# Why Are Developers Reluctant To Refactor Code?

- Even after buying into the advantages of refactoring, you might not do it !!!
  - You might not understand how to refactor
  - If the benefits are long term, why exert the effort now?
  - Refactoring code is an overhead activity
    - *"Refactoring is an overhead activity, I'm paid to write new, revenue-generating features"*
  - Refactoring might break existing programs

# Reducing the overhead of Refactoring

- Tools and technologies are available to allow refactoring to be done quickly and relatively painlessly
- Experienced Object-Oriented programmers suggest that overhead is compensated by time saved later on
- Might seem awkward at first, but starts feeling like an essential once you incorporate it into the cycle

# Sources

- These Slides are partly adopted from the lecture notes of Prof. Michael Godfrey on Object Oriented Programming
- Refactoring: Improving the Design of Existing Code, by Martin Fowler (et al.), 1999, Addison-Wesley
- [www.refactoring.com/catalog/index.html](http://www.refactoring.com/catalog/index.html)
- <https://refactoring.guru/>

# SOLID Principles

Presented by Mónica Rodrigues

January 24rd, 2017

# Contents

- What is SOLID
- Single responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

# What is SOLID?

S

SRP

Single  
Responsibility  
Principle

O

OCP

Open/Closed  
Principle

L

LSP

Liskovs  
Substitution  
Principle

I

ISP

Interface  
Segregation  
Principle

D

DIP

Dependency  
Inversion  
Principle

# Single Responsibility Principle

*“A class should have one and  
only one reason to change”*

# Single Responsibility Principle

```
6  public class Employee  
7  {  
8      public double CalculatePay(Money money)  
9      {  
10          //business logic for payment here  
11      }  
12  
13      public Employee Save(Employee employee)  
14      {  
15          //store employee here  
16      }  
17  }
```

Business logic

Persistence

There are **two** responsibilities



# Single Responsibility Principle

How to solve this?



# Single Responsibility Principle

```
21  public class Employee  
22  {  
23      public double CalculatePay(Money money)  
24      {  
25          //business logic for payment here  
26      }  
27  }  
28  
29  public class EmployeeRepository  
30  {  
31      public Employee Save(Employee employee)  
32      {  
33          //store employee here  
34      }  
35  }
```



Just create two different classes

## Open/Closed Principle

*“Software entities should be  
open for extension, but  
closed for modification.”*

# Open/Closed Principle

```
40  public enum PaymentType = { Cash, CreditCard };
41
42  public class PaymentManager
43  {
44      public PaymentType PaymentType { get; set; }
45
46      public void Pay(Money money)
47      {
48          if(PaymentType == PaymentType.Cash)
49          {
50              //some code here - pay with cash
51          }
52          else
53          {
54              //some code here - pay with credit card
55          }
56      }
57  }
```



Hum...and if I need to add a new payment type?

You need to modificate this class.

# Open/Closed Principle

open for extension

close for modification

```
60  public class Payment
61  {
62      public virtual void Pay(Money money)
63      {
64          // from base
65      }
66 }
```



```
68  public class CashPayment : Payment
69  {
70      public override void Pay(Money money)
71      {
72          //some code here - pay with cash
73      }
74 }
```

```
76  public class CreditCardPayment : Payment
77  {
78      public override void Pay(Money money)
79      {
80          //some code here - pay with credit card
81      }
82 }
```

# Liskov Substitution Principle

“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  $T$ ”

What do you say?



## Liskov Substitution Principle

*“A subclass should behave in such a way that it will not cause problems when used instead of the superclass.”*

# Liskov Substitution Principle

```
5   public class Employee
6   {
7       public virtual string GetProjectDetails(int employeeId)
8       {
9           Console.WriteLine("base project details");
10      }
11  }
13  public class CasualEmployee : Employee
14  {
15      public override string GetProjectDetails(int employeeId)
16      {
17          base.GetProjectDetails(employeeId);
18          Console.WriteLine("casual employee project details");
19      }
20  }
```



```
public class ContractualEmployee : Employee
{
    //broken your base class here
    public override string GetProjectDetails(int employeeId)
    {
        Console.WriteLine("contractual employee project details");
    }
}
```

# Liskov Substitution Principle

```
5  public class Employee
6  {
7      public virtual string GetProjectDetails(int employeeId)
8      {
9          Console.WriteLine("base project details");
10     }
11 }
```

```
13 public class CasualEmployee : Employee
14 {
15     public override string GetProjectDetails(int employeeId)
16     {
17         base.GetProjectDetails(employeeId);
18         Console.WriteLine("casual employee project details");
19     }
20 }
```

```
public class ContractualEmployee : Employee
{
    public override string GetProjectDetails(int employeeId)
    {
        base.GetProjectDetails(employeeId);
        Console.WriteLine("contractual employee project details");
    }
}
```



Much better

## Interface Segregation Principle

*“Clients should not be forced  
to depend upon interfaces  
that they don't use”*

# Interface Segregation Principle

```
62  public interface IEmployee
63  {
64      string GetProjectDetails(int employeeId);
65
66      string GetEmployeeDetails(int employeeId);
67  }
68
```

# Interface Segregation Principle

```
69  public class CasualEmployee : IEmployee  
70  {  
71      public string GetProjectDetails(int employeeId)  
72      {  
73          //code here - return base project details  
74      }  
75  
76      public string GetEmployeeDetails(int employeeId)  
77      {  
78          //code here - specific casual employee details  
79      }
```



WHY?????  
I don't need you!!

```
82  public class ContractualEmployee : IEmployee  
83  {  
84      public string GetProjectDetails(int employeeId)  
85      {  
86          //code here - specific project details  
87      }  
88  
89      public string GetEmployeeDetails(int employeeId)  
90      {  
91          throw new System.NotImplementedException();  
92      }  
93  }
```

# Interface Segregation Principle

How to solve this?



# Interface Segregation Principle

```
106  public interface IEmployee
107  {
108      string GetEmployeeDetails(int employeeId);
109  }
110
```



You need to create  
two interfaces

```
111  public interface IProject
112  [
113      string GetProjectDetails(int employeeId);
114  ]
115
```

# Interface Segregation Principle

```
116  public class CasualEmployee : IEmployee, IProject  
117  {  
118      public string GetEmployeeDetails(int employeeId)  
119      {  
120          //code here - specific casual employee details  
121      }  
122  
123      public string GetProjectDetails(int employeeId)  
124      {  
125          //code here - specific contractual employee details  
126      }  
127  }
```

```
129  public class ContractualEmployee : IProject  
130  {  
131      public string GetProjectDetails(int employeeId)  
132      {  
133          //code here - specific project details  
134      }  
135  }
```

# Dependency Inversion Principle

*“High-level modules should not depend on low-level modules. Both should depend on abstractions.”*

*“Abstractions should not depend upon details. Details should depend upon abstractions.”*

# Dependency Inversion Principle

```
175  public class Email  
176  {  
177      public void SendEmail()  
178      {  
179          // code to send mail  
180      }  
181  }
```

```
183  public class Notification  
184  {  
185      private Email _email;  
186      public void notification()  
187      {  
188          _email = new Email();  
189      }  
190  
191      public void PromotionalNotification()  
192      {  
193          _email.SendEmail();  
194      }  
195  }  
196
```

And if I need to send a  
notification by SMS?  
You need to change this.



# Dependency Inversion Principle

```
199  public interface IMessenger  
200  {  
201      void SendMessage();  
202  }
```

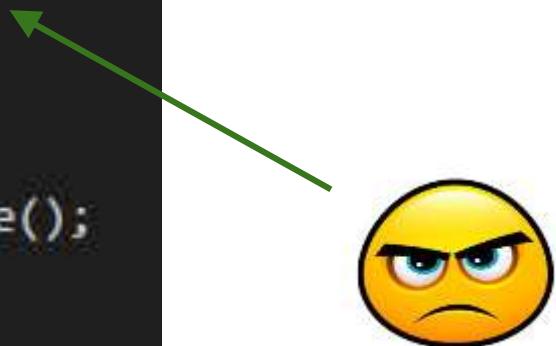
So, I create an interface and now?

```
204  public class Email : IMessenger  
205  {  
206      public void SendMessage()  
207      {  
208          // code to send email  
209      }  
210  }
```

```
212  public class SMS : IMessenger  
213  {  
214      public void SendMessage()  
215      {  
216          // code to send SMS  
217      }  
218  }
```

# Dependency Inversion Principle

```
220     public class Notification
221     {
222         private IMessenger _iMessenger;
223         public Notification()
224         {
225             _iMessenger = new Email();
226         }
227         public void DoNotify()
228         {
229             _iMessenger.SendMessage();
230         }
231     }
```



# Dependency Inversion Principle

Constructor injection:

```
235  public class Notification
236  {
237      private IMessenger _iMessenger;
238      public Notification(IMessenger pMessenger)
239      {
240          _iMessenger = pMessenger;
241      }
242      public void DoNotify()
243      {
244          _iMessenger.SendMessage();
245      }
246  }
```

# Dependency Inversion Principle

Property injection:

```
248  public class Notification
249  {
250      private IMessenger _iMessenger;
251
252      public IMessenger MessageService
253      {
254          private get;
255          set
256          {
257              _iMessenger = value;
258          }
259      }
260
261      public void DoNotify()
262      {
263          _iMessenger.SendMessage();
264      }
265  }
```

# Dependency Inversion Principle

Method injection:

```
268  public class Notification
269  {
270      public void DoNotify(IMessenger pMessenger)
271      {
272          pMessenger.SendMessage();
273      }
274  }
```

Keep in mind

**DRY - Don't repeat yourself**

+

**SLAP - Single layer abstraction principle**

+

**SOLID**

**BEST DEVELOPER**



Enjoy code and  
Keep it simple!



Thank you

# **S.O.L.I.D. Design Principles**

# SOLID Design Principles

- Introduced by Robert C. Martin
  - Agile Principles, Patterns, and Practices in C#
- **S**ingle Responsibility Principle
- **O**pen/Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**evelopment Inversion Principle

# Single Responsibility Principle (SRP)

- A class should have only one reason to change



# Single Responsibility Principle (SRP)

- A class should have *only one reason to change*
- A responsibility is “**a reason for change.**”
- The responsibilities of a class are axes of change.
  - If it has two responsibilities, they are **coupled** in the design, and so have to change together.
- If a class has **more than one** responsibility, it may become **fragile** when any of the requirements change

# SRP – Example

Report
GetData()
FormatReport()
Print()

```
namespace Dimccasta.SOLID
{
    class Report
    {
        private IList<ReportDataElement> GetData()
        {
            Console.WriteLine("\nGetting Data...");
            return new List<ReportDataElement>() { new ReportDataElement("Report 1") };
        }

        private void FormatReport()
        {
            Console.WriteLine("\nFormatting Report...");
        }

        public void Print()
        {
            GetData();
            FormatReport();
            Console.WriteLine("\nPrinting Report...");
        }
    }
}
```

## SRP – Example *cnt.*

```
public class DataAccess
{
    public IList<ReportDataElement> GetData()
    {
        Console.WriteLine("\nGetting Data...");
        return new List<ReportDataElement>() { new ReportDataElement() };
    }
}

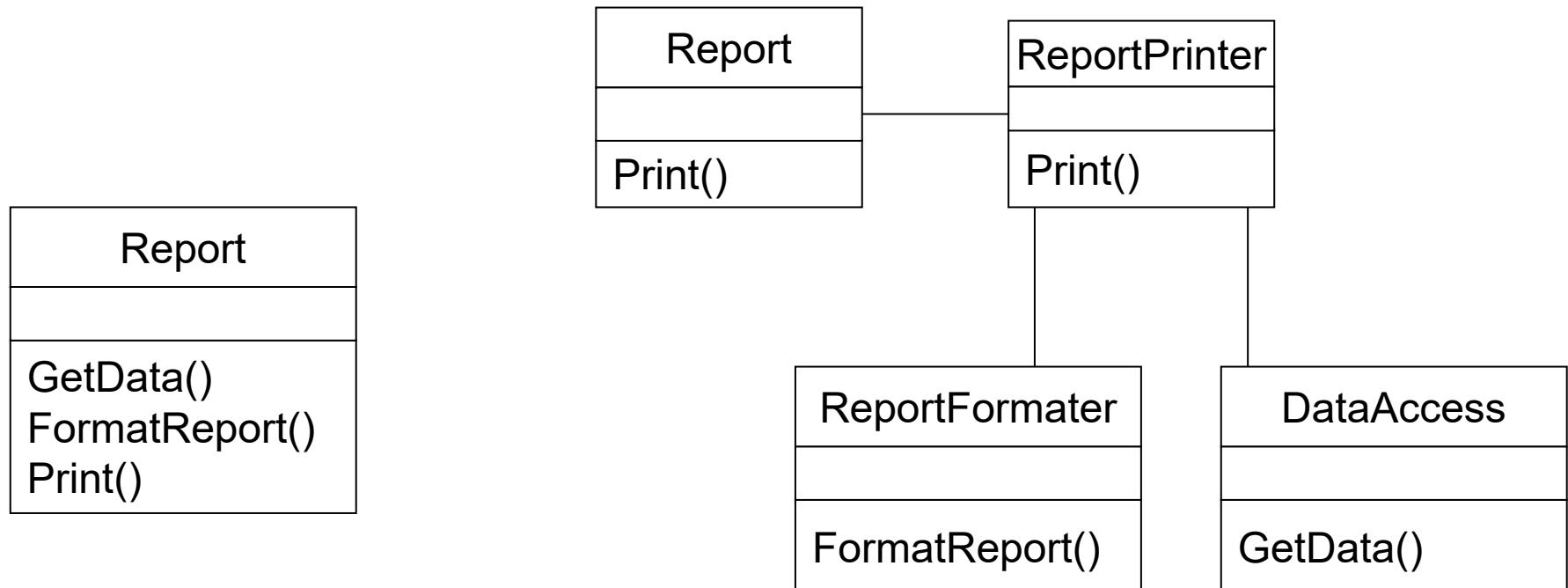
public class ReportFormatter
{
    public void FormatReport()
    {
        Console.WriteLine("\nFormatting Report...");
    }
}

public class ReportPrinter
{
    public void Print()
    {
        DataAccess dataAccess = new DataAccess();
        ReportFormatter reportFormatter = new ReportFormatter();
        dataAccess.GetData();
        reportFormatter.FormatReport();
        Console.WriteLine("\nPrinting Report...");
    }
}

class Report
{
    public void Print()
    {
        ReportPrinter reportPrinter = new ReportPrinter();
        reportPrinter.Print();
    }
}
```



# SRP – Example *cnt.*



# Benefits of SRP

- code is smaller
  - easier to read
  - easier to understand
  - easier to maintain
- Code is potentially easier to test
- Change is easier to manage
  - code is easier to replace

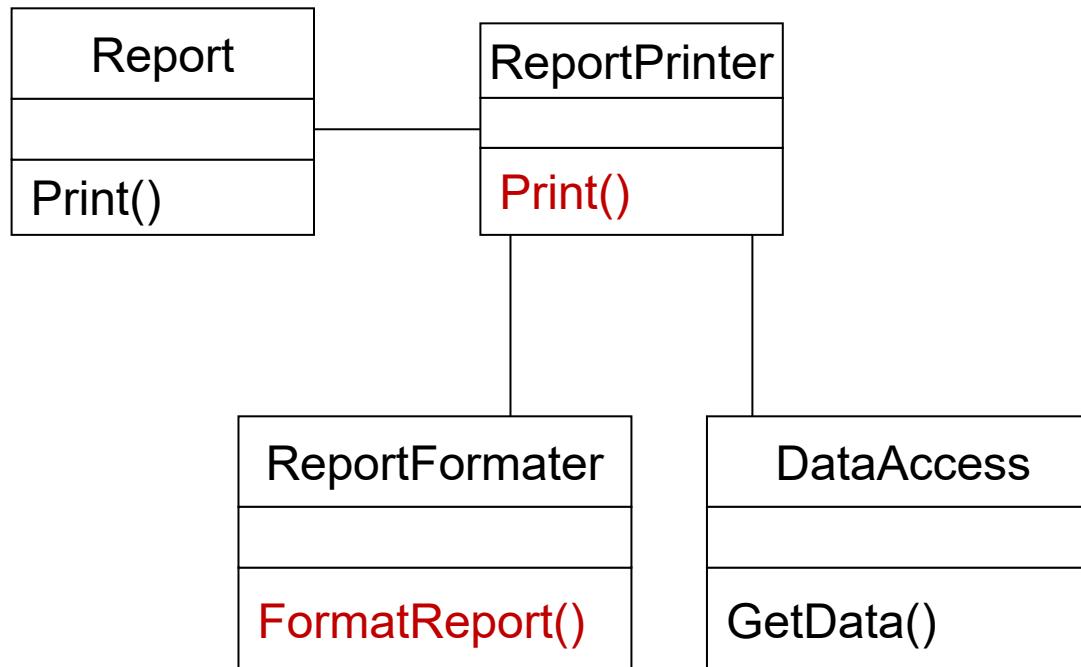
# Open/Closed Principle (OCP)

Software entities (module, classes, functions, etc.) should be **open for extension** but **closed for modification**

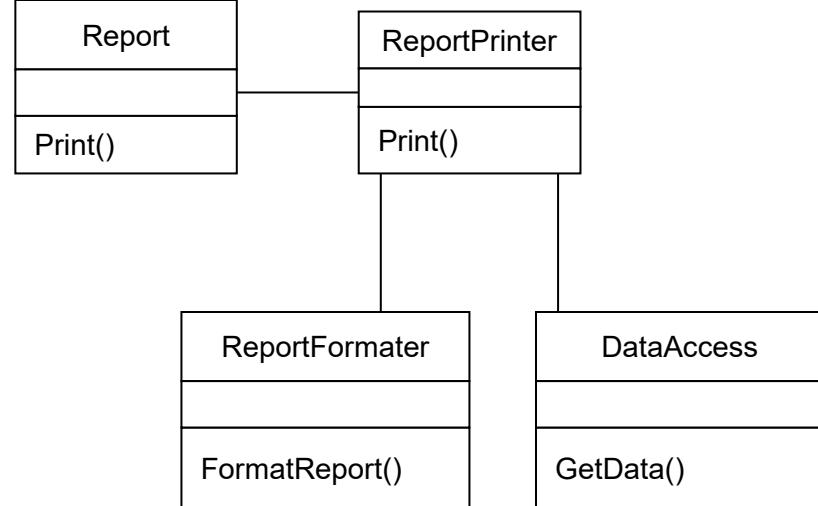
- Classes should be written so that they can be extended without requiring modification.
  - extend the behaviour of a system (in response to a requested change)
  - add new code
  - But not modifying the existing code
- Once completed, the implementation of a class should only be modified to correct errors; new or changed features would require that a different class be created
- **Mechanism?**
  - **Abstraction** and **subtype polymorphism** are the keys to the OCP
    - Introducing Abstract Base Classes
    - Implementing common Interfaces
    - Deriving from common interfaces or classes

# OCP – Example

- Changing the print code
  - Format the report into tabloid (instead of 8-1/2 X 11)
  - Print the report to a dot-matrix (instead of laser) printer



# OCP – Example *ct.*



```
public class ReportFormatter
{
    public virtual void FormatReport()
    {
        Console.WriteLine("\nFormatting Report for 8-1/2 X 11...");
    }
}

public class TabloidReportFormatter : ReportFormatter
{
    public override void FormatReport()
    {
        Console.WriteLine("\nFormatting Report for 11 X 17...");
        //base.FormatReport();
    }
}
```

# OCP – Example *cnt.*

```
public class ReportPrinter
{
    public void Print()
    {
        DataAccess dataAccess = new DataAccess();
        ReportFormatter reportFormatter = new ReportFormatter();
        dataAccess.GetData();
        reportFormatter.FormatReport();
        Console.WriteLine("\nPrinting Report to laser printer...");
    }
}

public class TabloidReportPrinter : ReportPrinter
{
    public override void Print()
    {
        DataAccess dataAccess = new DataAccess();
        ReportFormatter reportFormatter = new TabloidReportFormatter();
        dataAccess.GetData();
        reportFormatter.FormatReport();
        Console.WriteLine("\nPrinting Report to dot-matrix printer...");
        //base.Print();
    }
}
```

# OCP – Example *cnt.*

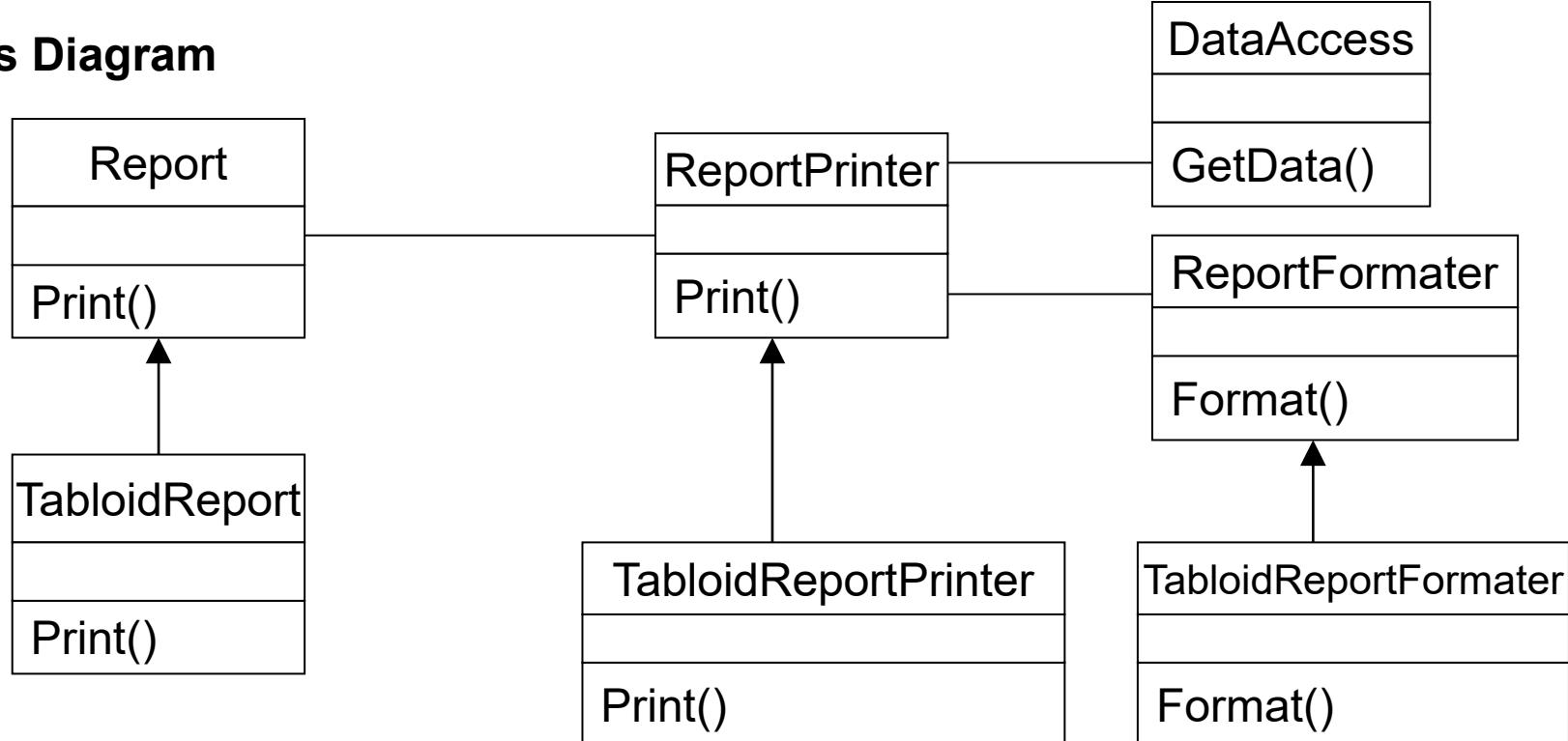
```
class Report
{
    public virtual void Print()
    {
        ReportPrinter reportPrinter = new ReportPrinter();
        reportPrinter.Print();
    }
}

public class TabloidReport : Report
{
    public override void Print()
    {
        ReportPrinter reportPrinter = new TabloidReportPrinter();
        reportPrinter.Print();|
        //base.Print();
    }
}
```

**Note the extension of the system behaviour without making modification to existing code** 13/ 39

# OCP – Example *cnt.*

UML Class Diagram

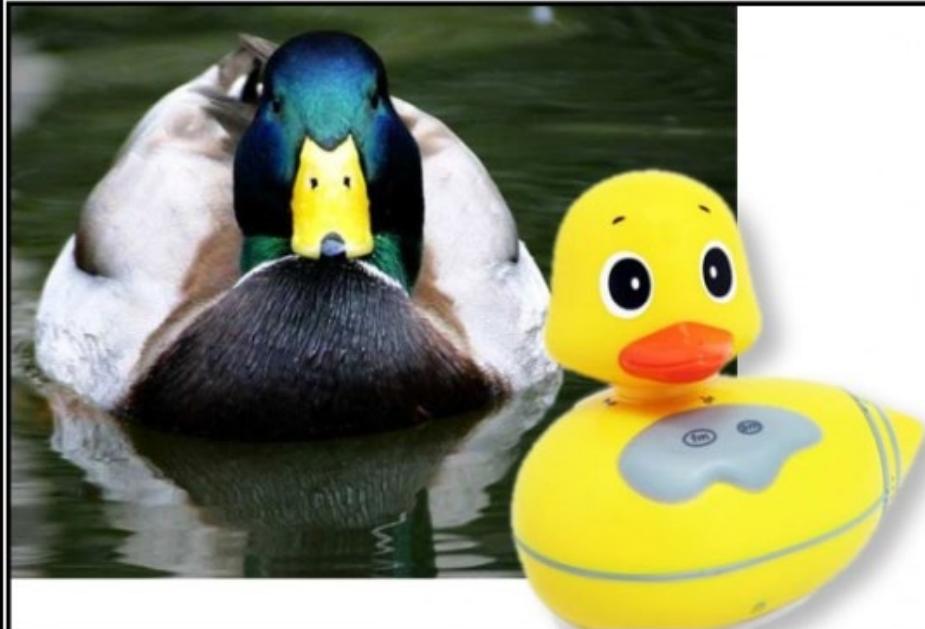


# Benefits of OCP

- design is more stable
  - existing (working) code does not change
  - changes are made by adding, not modifying, code
  - changes are isolated and do not cascade throughout the code
- code is potentially easier to test
- change is easier to manage
  - code is easier to replace
  - design is extensible
- code is potentially reusable
- Higher abstraction

# Liskov Substitution Principle (LSP)

*Subclasses should be substitutable for their base classes*



LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You  
Probably Have The Wrong Abstraction

# Liskov Substitution Principle (LSP)

- *Subclasses should be substitutable for their base classes*
- LSP states that if a program module is using a Base class, then the reference to the Base class can be replaced with a Derived class without affecting the functionality of the program module.
- “A derived class should have some kind of specialized behaviour (it should provide the same services as the superclass, only some, at least, are provided differently.)”
- The **contract** of the base class *must* be honoured by the derived class.
- If this principle is violated, then it will cause the Open-Closed Principle to be violated also. *Why?*

# LSP – Benefits

- design is more flexible
  - a base type can be confidently substituted for a derived type
- code is potentially easier to test
- required to support the Open/Closed Principle

# Interface Segregation Principle (ISP)

- *Many client-specific interfaces are better than one general purpose interface*
- *Clients should not depend on interfaces that they do not use*
  - Interfaces should be thin and not fat
  - E.g. if you have an interface with 20 methods, then not all implementers of that interface might implement all of those methods
    - Bring large and fat interfaces into smaller ones which classify the related methods
- If you have a class with several different uses, create separate (narrow) interfaces for each use.
- Why ISP?
  - The purpose is to make clients use as small and as **coherent** an interface as possible.
  - Fat interfaces lead to inadvertent **couplings** and accidental dependencies between classes.

## ISP – Example

```
internal interface IDataAccess
{
    void SaveData(DataElement dataElement);
    IList<DataElement> QueryData(string criteria);
    IList<DataElement> GetReportData();
    ...
}
```



```
internal abstract class DataAccess : IDataAccess
{
    public abstract void SaveData(DataElement dataElement);
    public abstract IList<DataElement> QueryData(string criteria);
    public abstract IList<DataElement> GetReportData();
}

class Report.DataAccess : DataAccess
{
    ...
    public override IList<DataElement> GetReportData()
    {
        Console.WriteLine("\nGetting Data...");
        return new List<DataElement>() { new DataElement() { CustomerFir
    }

    public override IList<DataElement> QueryData(string criteria)
    {
        throw new NotImplementedException();
    }

    public override void SaveData(DataElement dataElement)
    {
        throw new NotImplementedException();
    }
}
```

## ISP – Example *cnt.*

Breaking the fat  
interface of  
IDataAccess into  
two smaller ones:

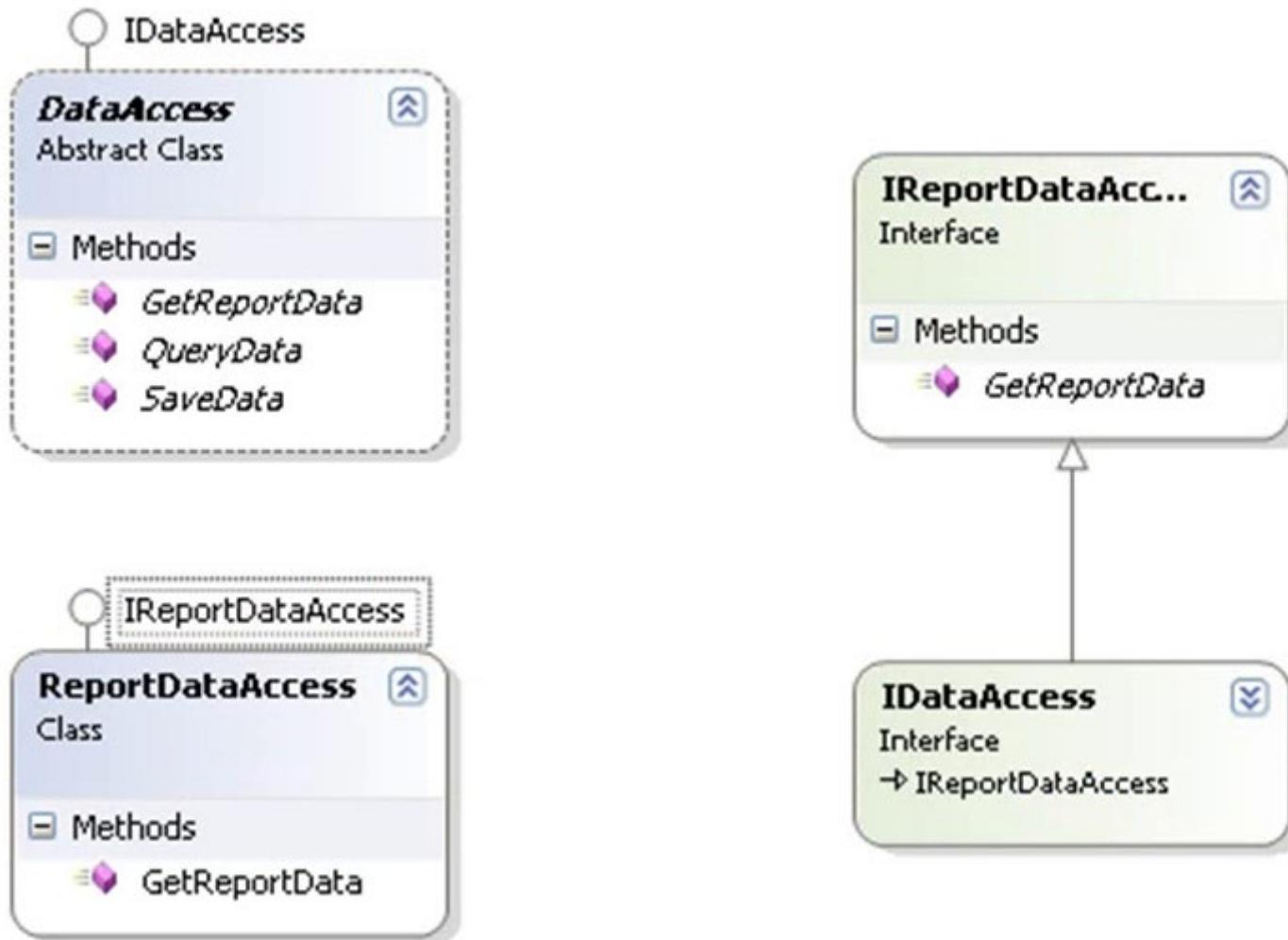
```
internal interface IDataAccess : IReportDataAccess
{
    void SaveData(DataElement dataElement);
    IList<DataElement> QueryData(string criteria);
}

public interface IReportDataAccess
{
    IList<DataElement> GetReportData();
}

class ReportDataAccess : IReportDataAccess
{
    public override IList<DataElement> GetReportData()
    {
        Console.WriteLine("\nGetting Data...");
        return new List<DataElement>() { new DataElement() { CustomerF
    }
}
```

Class that implements the interface does no longer need to implement all methods of the class which it does not need

# ISP – Example *cnt.*



# ISP – Benefits

- Cohesion is increased
  - clients can demand cohesive interfaces
- Design is more stable
  - changes are isolated and do not cascade throughout the code
- Supports the Liskov Substitution Principle (*Why?*)

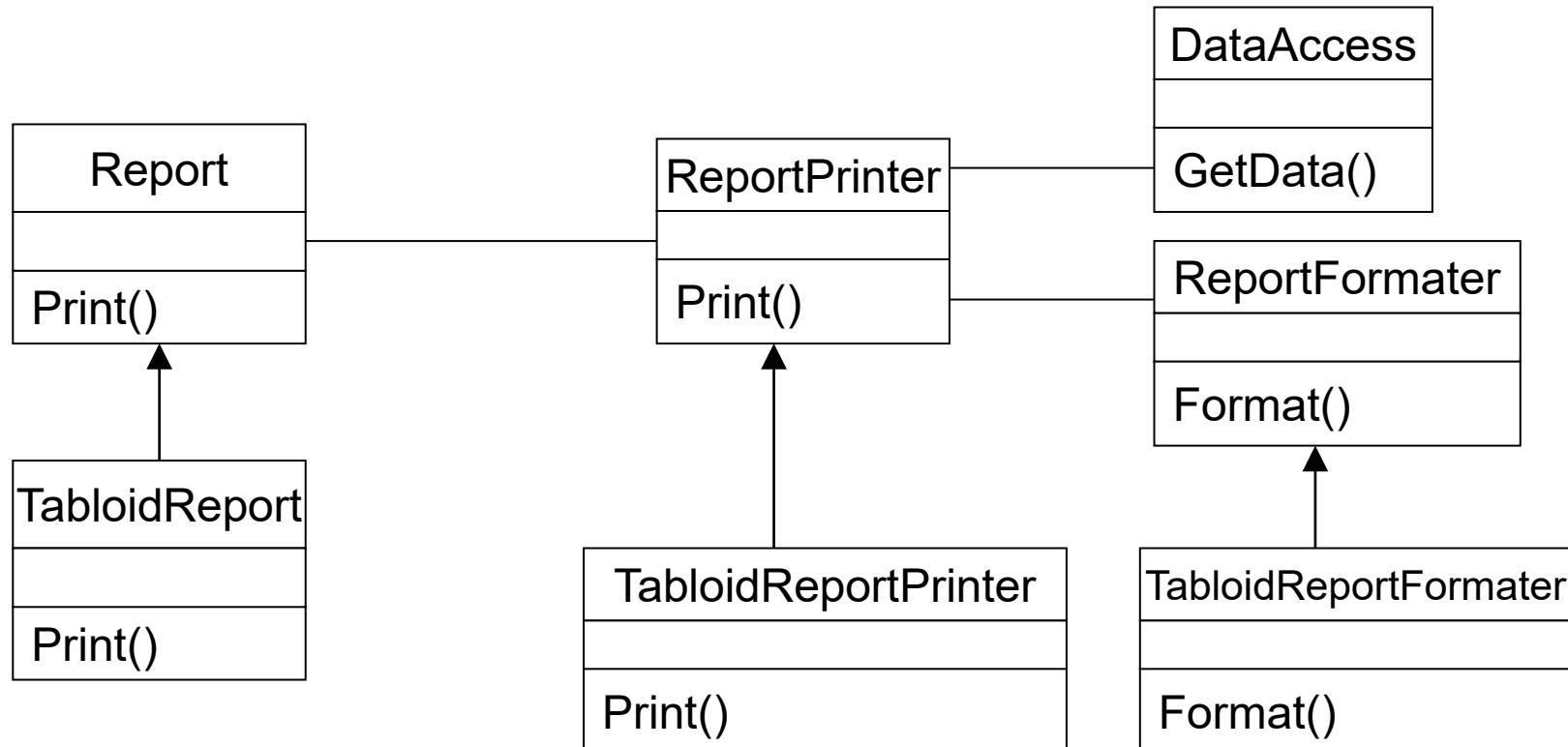
# Dependency Inversion Principle (DIP)

*High level modules should not depend on low level modules*

- *both should depend upon abstractions*
  - *Dependency upon lower-level components limits the reuse opportunities of the higher-level components*
- *Abstractions should not depend on details*
- Change the relation between concrete classes to relationships among abstractions
  - Abstraction mechanism: Abstract base classes or Interfaces
- The goal of the dependency inversion principle is to
  - decouple high-level components from low-level components
  - such that reuse with different low-level component implementations becomes possible
- High level classes should contain the “business logic”, low level classes should contain the details of the (current) implementation.
- Access instances using interfaces or abstract classes to avoid dependency inversion.

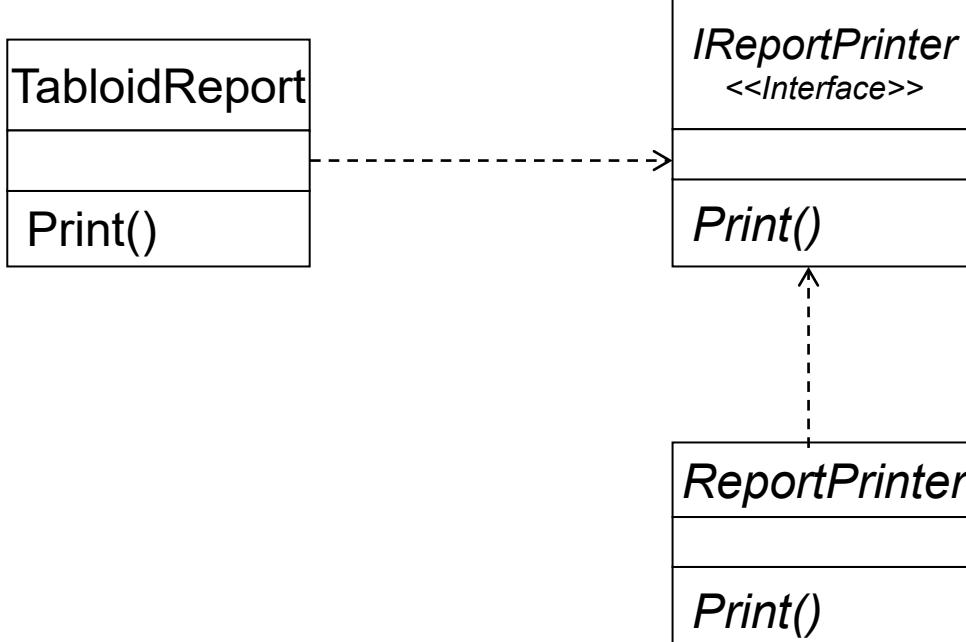
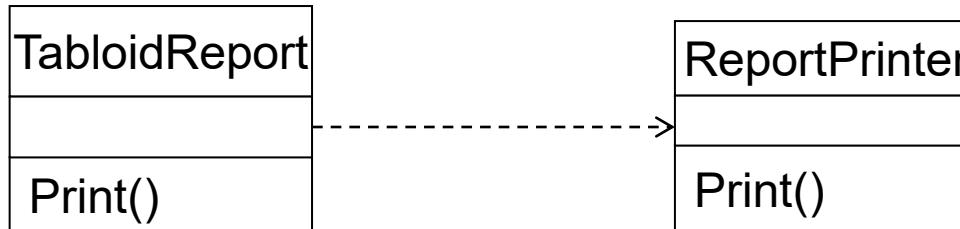


# DIP – Example



# DIP

```
internal class TabloidReport : Report
{
    public override void Print()
    {
        ReportPrinter reportPrinter = new TabloidReportPrinter();
        reportPrinter.Print();
        //base.Print();
    }
}
```



# DIP

```
internal class TabloidReport : Report
{
    public override void Print()
    {
        ReportPrinter reportPrinter = new TabloidReportPrinter();
        reportPrinter.Print();
        //base.Print();
    }
}
```

```
internal interface IReportPrinter
{
    void Print();
}

abstract class ReportPrinter : IReportPrinter
{
    public abstract void Print();
}
```

```
internal class TabloidReport : Report
{
    public override void Print()
    {
        IReportPrinter reportPrinter = new TabloidReportPrinter();
        reportPrinter.Print();
        //base.Print();
    }
}
```

# DIP – Beefits

- Enables *design by contract*
- change is easier to manage
  - code is easier to replace
  - design is extensible
- code is potentially easier to test

# Design Practice 1

You have this class for controlling key functionalities of a basic coffee machine

 **BasicCoffeeMachine**

- Map<CoffeeSelection, Configuration> configMap
- Map<CoffeeSelection, GroundCoffee> groundCoffee
- BrewingUnit brewingUnit

◆ + BasicCoffeeMachine(Map<CoffeeSelection, GroundCoffee> coffee)

● + Coffee brewCoffee(CoffeeSelection selection)

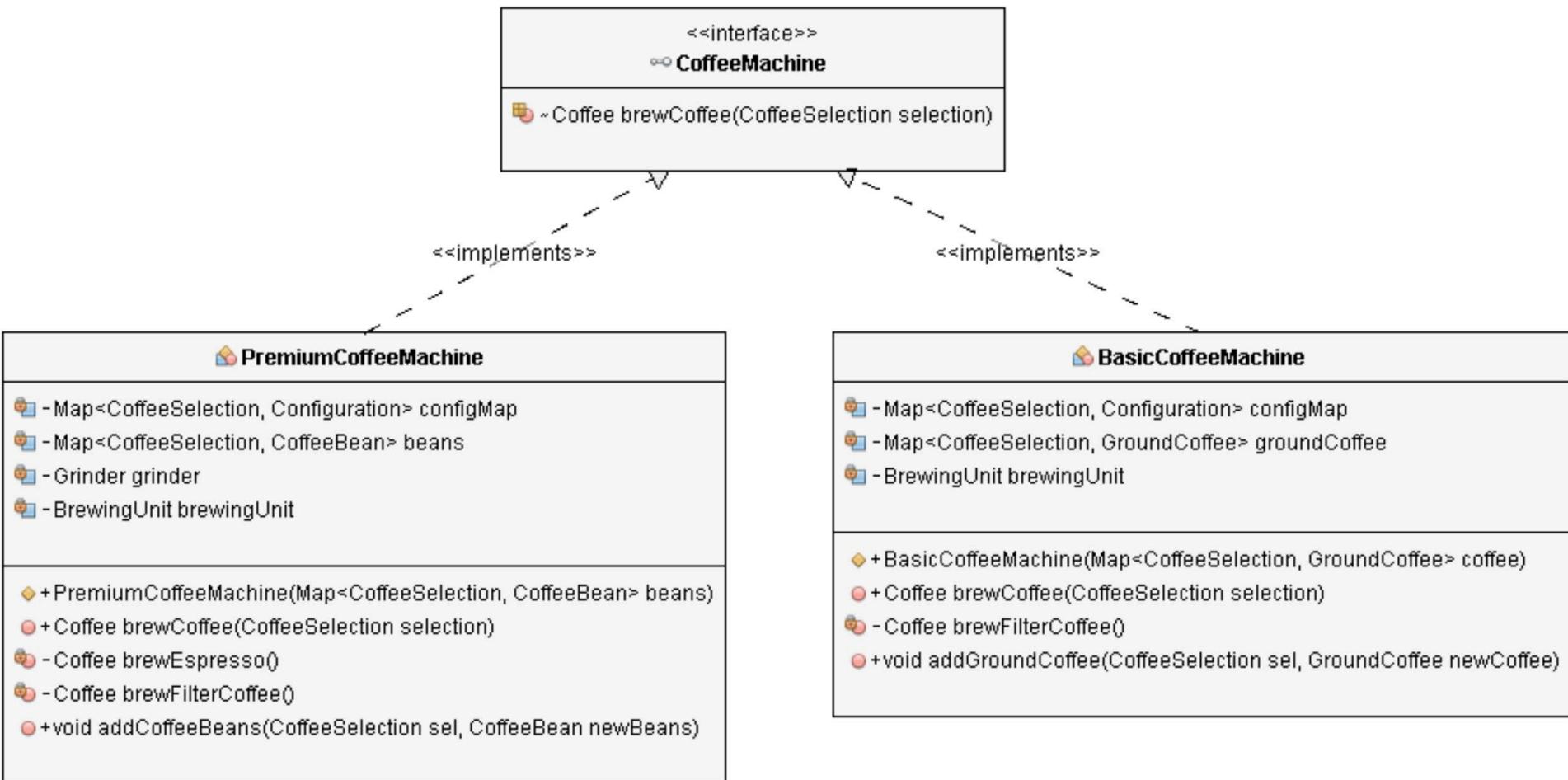
● - Coffee brewFilterCoffee()

● + void addGroundCoffee(CoffeeSelection sel, GroundCoffee newCoffee)



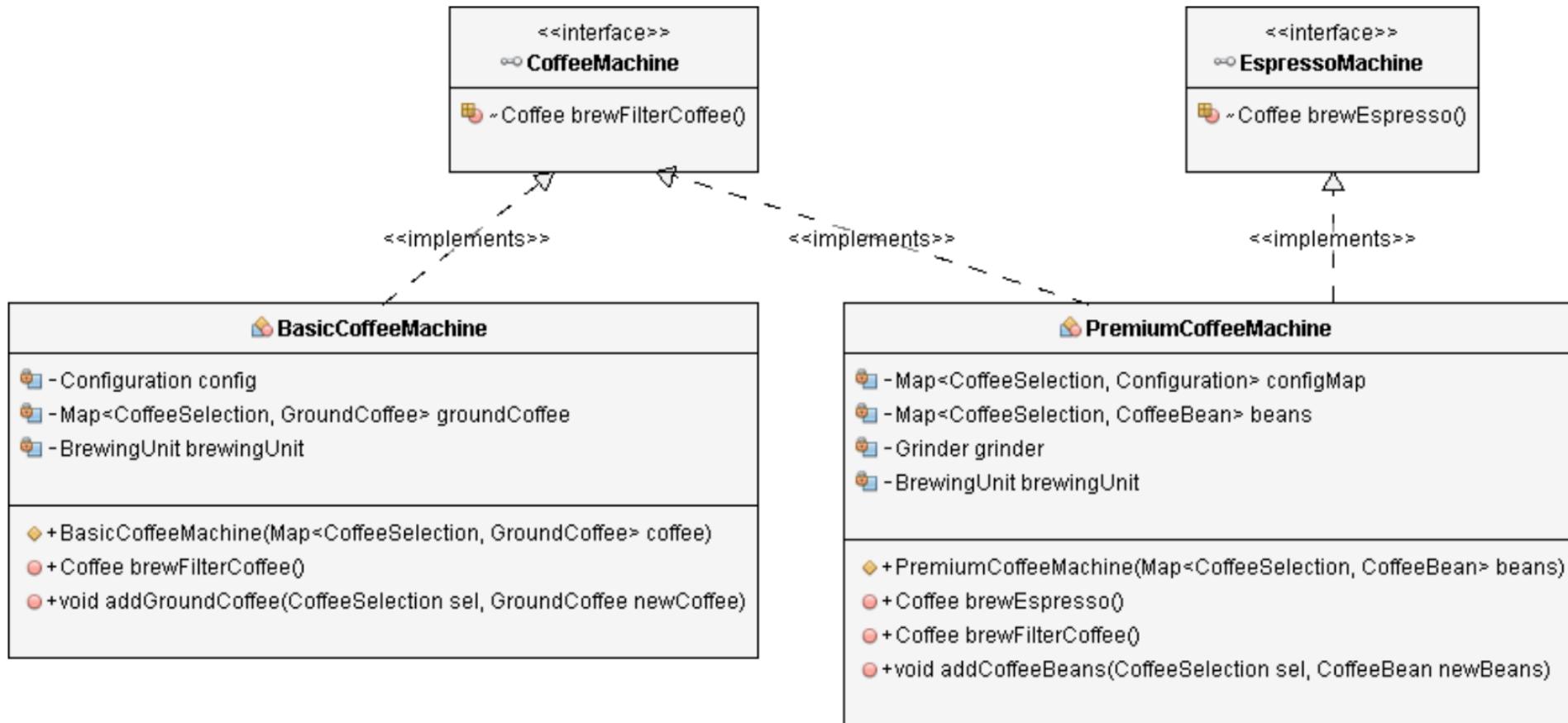
Now your company has made an Advanced Coffee machine, which has a Grinder and Espresso maker as well. How would you change the code of BasicCoffeeMachine to make a controller for the advanced machine?

# Design Practice 1



This design is based on Open/Close Principle

# Design Practice 1



This design applies Dependency Inversion Principle

# Which SOLID design principle is violated in this code?

```
public class Book {  
  
    private String name;  
    private String author;  
    private String text;  
  
    //constructor, getters and setters  
  
    // methods that directly relate to the book properties  
    public String replaceWordInText(String word){  
        return text.replaceAll(word, text);  
    }  
  
    public boolean isWordInText(String word){  
        return text.contains(word);  
    }  
  
    .  
    .  
    .  
  
    void printTextToConsole(){  
        // our code for formatting and printing the text  
    }  
}
```

# Which SOLID design principle is violated in this code?

```
1 public interface Car {  
2  
3     void turnOnEngine();  
4     void accelerate();  
5 }  
  
1 public class MotorCar implements Car { 1 public class ElectricCar implements Car {  
2  
3     private Engine engine;  
4  
5     //Constructors, getters + setters  
6  
7     public void turnOnEngine() { 3     public void turnOnEngine() {  
8         //turn on the engine!  
9         engine.on();  
10    } 5     }  
11  
12    public void accelerate() { 6  
13        //move forward!  
14        engine.powerOn(1000);  
15    } 7     public void accelerate() {  
16 } 8         //this acceleration is crazy!  
9     } 10    }
```

# How do you improve this code?

```
class Customer
{
    void Add(Database db)
    {
        try
        {
            db.Add();
        }
        catch (Exception ex)
        {
            File.WriteAllText(@"C:\Error.txt", ex.ToString());
        }
    }
}
```

# Preserve SRP

```
class Customer
{
    private FileLogger logger = new FileLogger();
    void Add(Database db)
    {
        try {
            db.Add();
        }
        catch (Exception ex)
        {
            logger.Handle(ex.ToString());
        }
    }
}
class FileLogger
{
    void Handle(string error)
    {
        File.WriteAllText(@"C:\Error.txt", error);
    }
}
```

# Question?

- Imagine you use an external library which contains a class Car. The Car has a method `brake`. In its base implementation, this method only slows down the car but you also want to turn on the brake lights.
- How do you extend the Car's behavior without touching the original class from the library?
- You would create a subclass of Car and override the method `brake`. After calling the original method of the super class, you can call your own `turnOnBrakeLights` method.

# Question?

- In an adventure game, you have a class for your main character. The player can either be a warrior or an archer or a wizard. Instead of a class which can perform all the actions, like strike, shoot and heal, you would create three different classes, one for each character type. In the end, you would not only have a Character class but also a Warrior, an Archer and a Wizard, all inheriting from Character and implementing their specific actions.
- Which SOLID principle is respected here?
- Interface Segregation

Lecture 10

# Design Patterns

# What is a Design Pattern?

- A design pattern is a **general reusable solution** to a commonly **occurring problem** in software design.
  - It is not a finished design that can be transformed directly into code.
  - It is a description or **template** for how to solve a problem that can be used in many different situations.
- Object-oriented design patterns typically show **relationships and interactions between classes** or objects, without specifying the final application classes or objects that are involved.

# Classification

- Creational
  - Deal with object **creation** mechanisms
- Structural
  - Ease the design by identifying a simple way to realize relationships between entities
- Behavioural
  - Identify **common communication patterns** between objects; aimed at increasing flexibility in carrying out this communication.
- Architectural
  - Are used for software architecture

# Classification

Creational	Structural	Behavioural	Architecture
Factory method Abstract Factory Builder Lazy Loading Object pool Prototype Singleton Multiton Resource acquisition is initialization	Adapter Bridge Composite Decorator Façade Flyweight Proxy	Null Object Null Object Command Interpreter Iterator Mediator Memento Observer State Chain of responsibility Strategy Specification Template method Visitor	Layers Presentation- abstraction-control Three-tier Pipeline Implicit invocation Blackboard system Peer-to-peer <b>Model-View-Controller</b> Service-oriented architecture Naked objects

# Outline

- Creational Patterns
  - Singleton
  - Lazy Loading
  - Factory Method
  - Abstract Factory
- Structural Patterns
  - Adapter
  - Decorator
- Behavioural
  - Iterator
  - Mediator
  - Memento
  - Strategy
  - Command
- Architectural
  - MVC
- Anti patterns

# Singleton

- Restricts the instantiation of a class to one "single" instance.
- This is useful when exactly one object is needed to coordinate actions across the system.
- The key idea in this pattern is to make the class itself responsible for controlling its instantiation

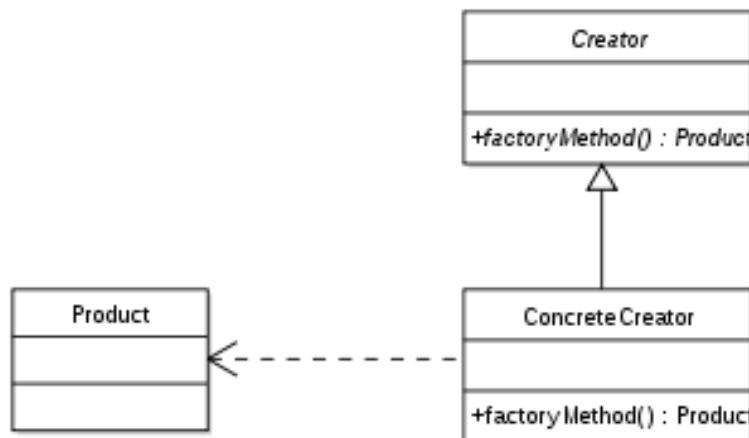
# Singleton

```
public final class Singleton {  
  
    private static final Singleton INSTANCE = new Singleton();  
  
    private Singleton() {}  
  
    public static Singleton getInstance() {  
        return INSTANCE;  
    }  
}
```

# Factory Method

Creational DP

- **Intent**
  - Implements the concept of factory
    - Defines an **interface for creating an object**, but let **client** decide which class to instantiate.
    - The Factory method lets a class defer instantiation to clients
- **Problem**
  - A framework needs to standardize the architectural model for a range of applications, but **allow for individual applications to define their own domain objects** and provide for their instantiation.
- **Structure**



# Factory Method

Creational DP

```
public interface ImageReader {  
    public DecodedImage getDecodedImage();  
}  
  
public class GifReader implements ImageReader {  
    public DecodedImage getDecodedImage() {  
        // ...  
        return decodedImage;  
    }  
}  
  
public class JpegReader implements ImageReader {  
    public DecodedImage getDecodedImage() {  
        // ...  
        return decodedImage;  
    }  
}
```

```
public class ImageReaderFactory {  
    public static ImageReader  
        getImageReader(InputStream is) {  
        int imageType = determineImageType(is);  
        switch (imageType) {  
            case ImageReaderFactory.GIF:  
                return new GifReader(is);  
            case ImageReaderFactory.JPG:  
                return new JpegReader(is);  
            // etc.  
        }  
    }  
}
```

# Factory Method

Creational DP

```
//Empty vocabulary of actual object
public interface IPerson
{
    string GetName();
}

public class Villager : IPerson
{
    public string GetName()
    {
        return "Village Person";
    }
}

public class CityPerson : IPerson
{
    public string GetName()
    {
        return "City Person";
    }
}

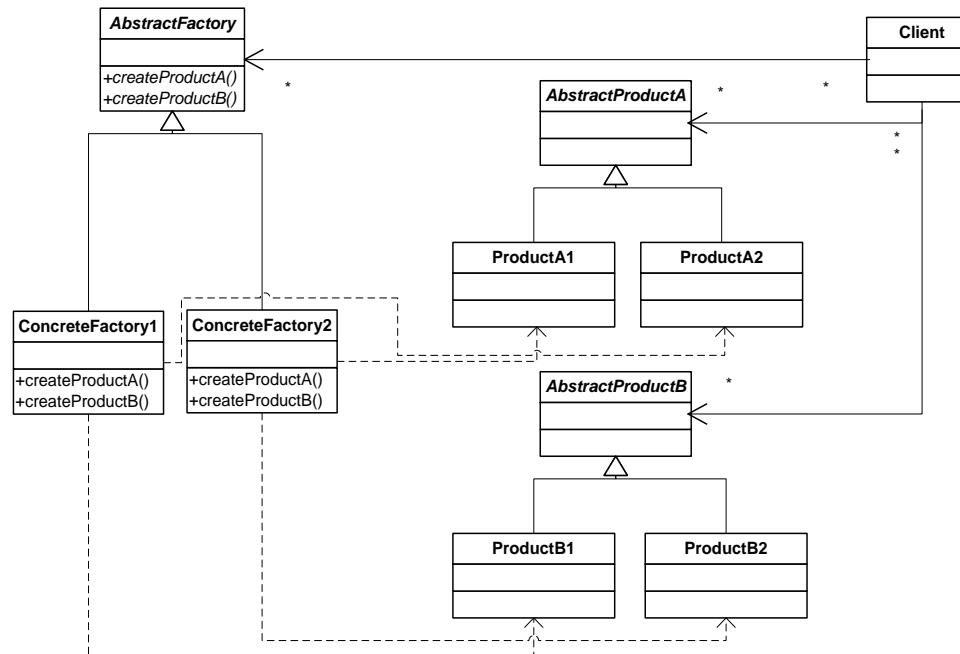
public enum PersonType
{
    Rural,
    Urban
}
```

```
/// <summary>
/// Implementation of Factory - Used to create objects
/// </summary>
public class Factory
{
    public IPerson GetPerson(PersonType type)
    {
        switch (type)
        {
            case PersonType.Rural:
                return new Villager();
            case PersonType.Urban:
                return new CityPerson();
            default:
                throw new NotSupportedException();
        }
    }
}
```

# Abstract Factory

Creational DP

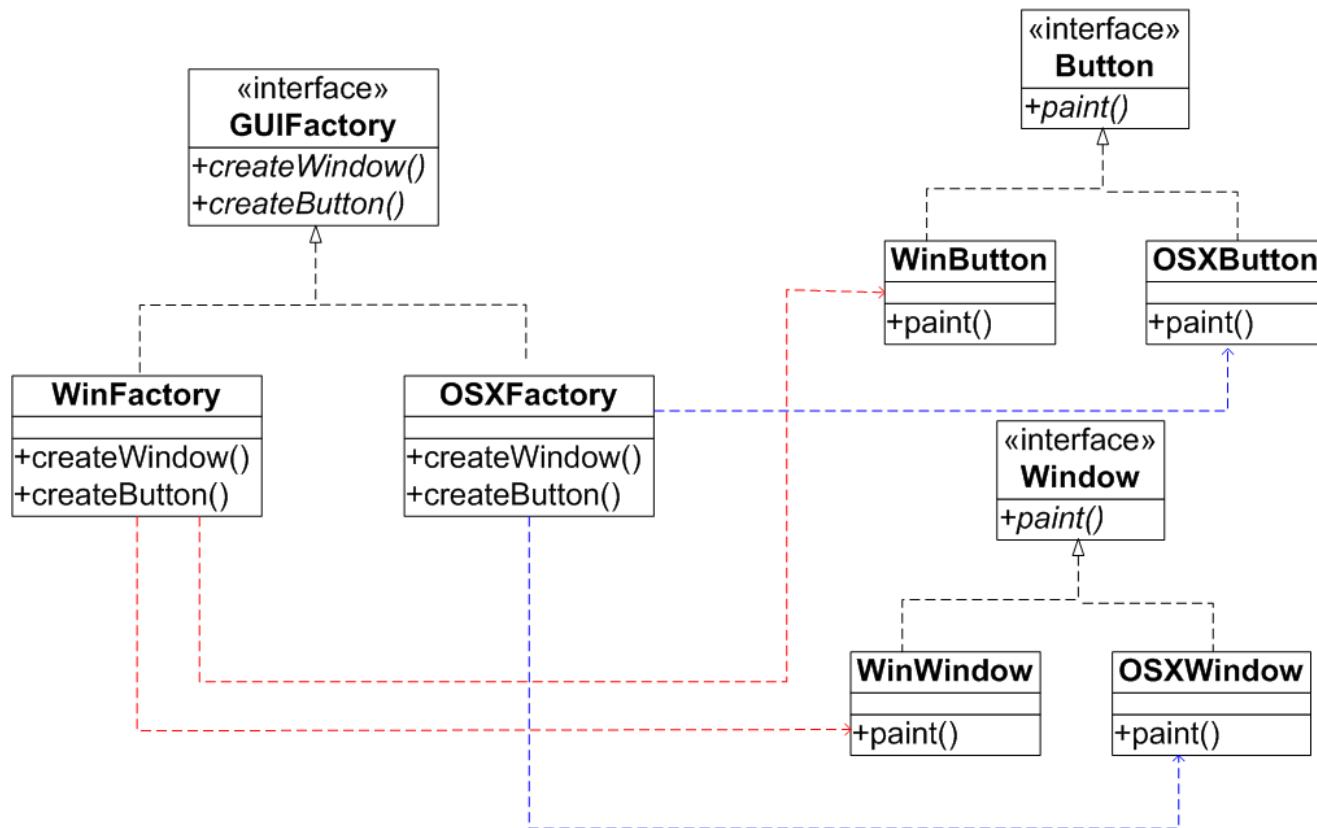
- Intent
  - Provide an interface for creating **families** of related or dependent objects **without specifying their concrete classes**.
- Problem
  - When **clients** cannot anticipate groups of classes to instantiate
  - If an application is to be **portable**, it needs to encapsulate platform dependencies. These "**platforms**" might include: windowing system, operating system, database, etc.



# Abstract Factory

Creational DP

- Design a portable GUI system which paints windows and buttons on different operating systems.



# Abstract Factory

Creational DP

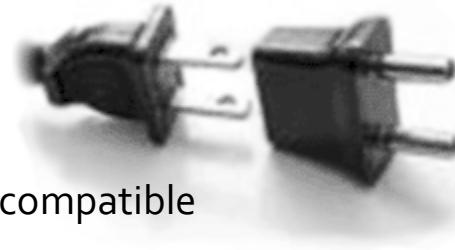
```
interface GUIFactory {  
    public Button createWindow();  
    public Button createButton();  
}  
  
class WinFactory implements GUIFactory {  
    public Button createButton() {  
        return new WinButton();  
    }  
  
    public Button createWindow() {  
        return new WinWindow();  
    }  
}  
  
class OSXFactory implements GUIFactory {  
    public Button createButton() {  
        return new OSXButton();  
    }  
  
    public Button createWindow() {  
        return new OSXWindow();  
    }  
}  
  
interface Button {  
    public void paint();  
}  
  
class WinButton implements Button {  
    public void paint() {  
        // ...  
    }  
}  
  
class OSXButton implements Button {  
    public void paint() {  
        // ....  
    }  
}  
  
interface Window {  
    public void paint();  
}  
  
class WinWindow implements Window {  
    // ...  
}  
  
class OSXWindow implements Window {  
    //..  
}
```

# Outline

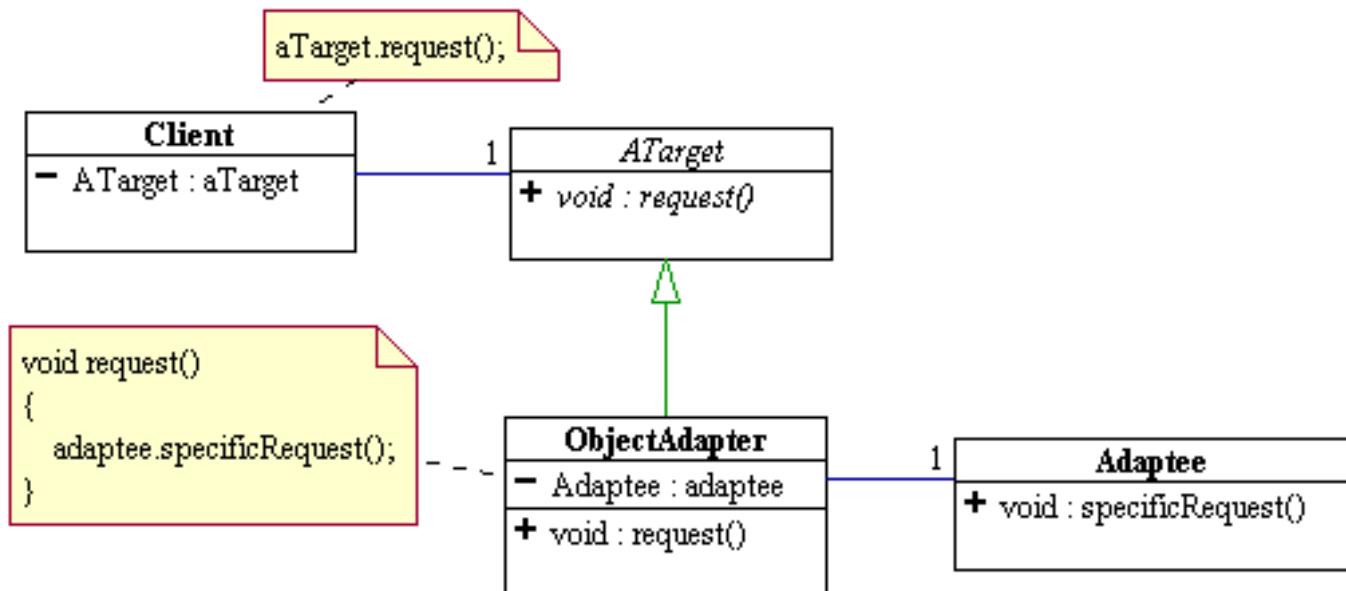
- Creational Patterns
  - Singleton
  - Lazy Loading
  - Factory Method
  - Abstract Factory
- Structural Patterns
  - Adapter
  - Decorator
- Behavioural
  - Iterator
  - Mediator
  - Memento
  - Strategy
  - Command
- Architectural
  - MVC
- Anti patterns

# Adapter

Structural DP



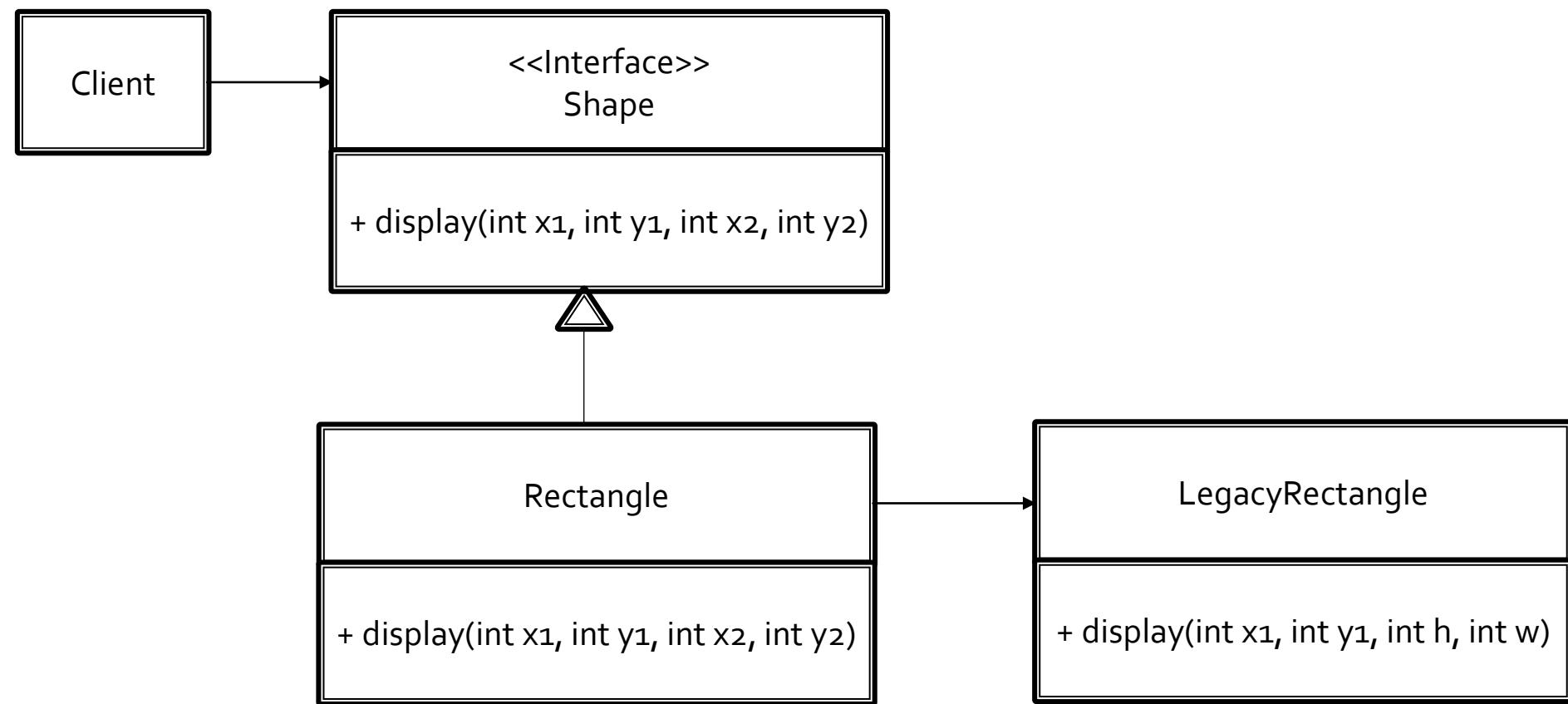
- Intent
  - Convert the interface of a class into another interface clients expect.
  - Wrap an existing class with a new interface.
  - Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Problem
  - An “off the shelf” component offers compelling functionality that you would like to reuse, but its “view of the world” is not compatible with the design of the system currently being developed.
  - Reuse of legacy components



# Adapter

Structural DP

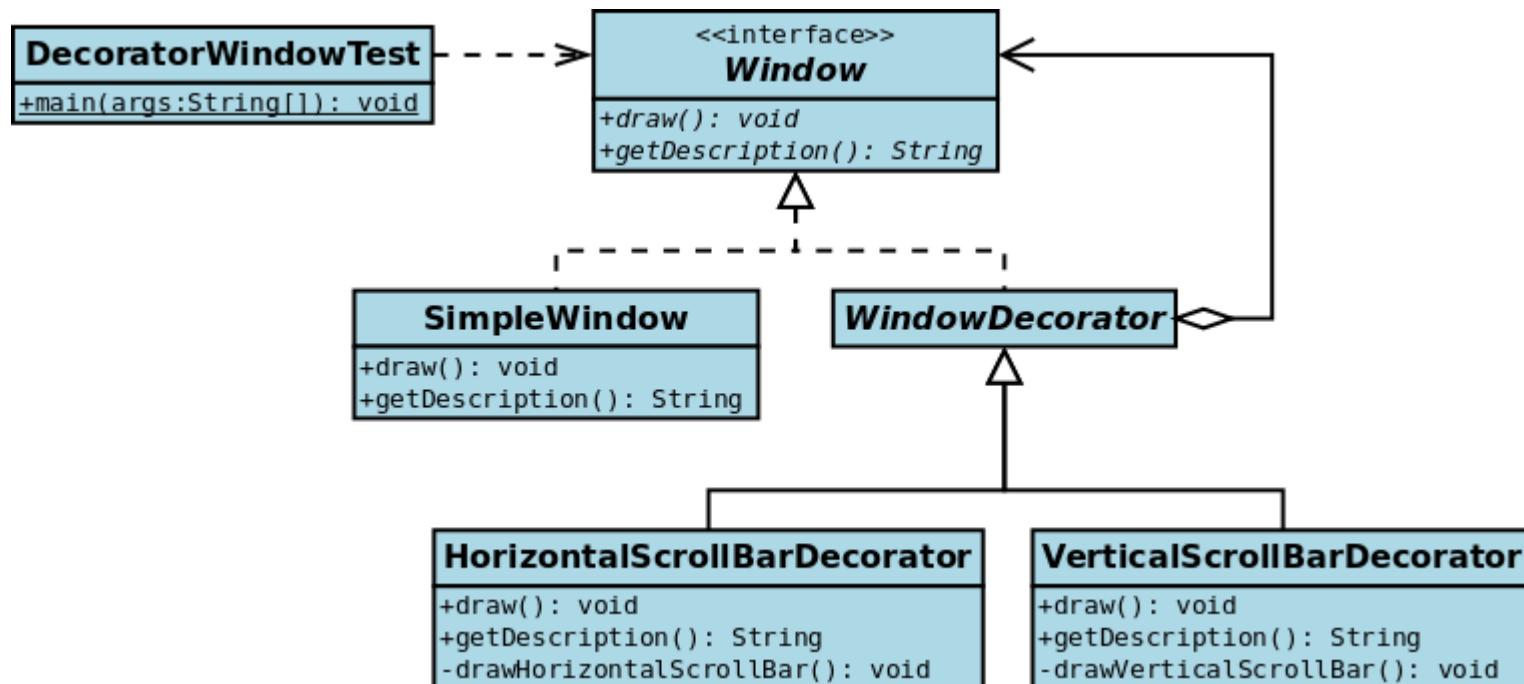
a legacy Rectangle component's display() method expects to receive "x, y, w, h" parameters. But the client wants to pass "upper left x and y" and "lower right x and y". This incongruity can be reconciled by adding an additional level of indirection – i.e. an Adapter object.



# Decorator

Structural DP

- Intent
  - Adding additional functionality to a particular **object** as opposed to a class of objects
- Problem
  - You want to add behavior or state to individual objects **at run-time**. Inheritance is not feasible because it is static and applies to an entire class.



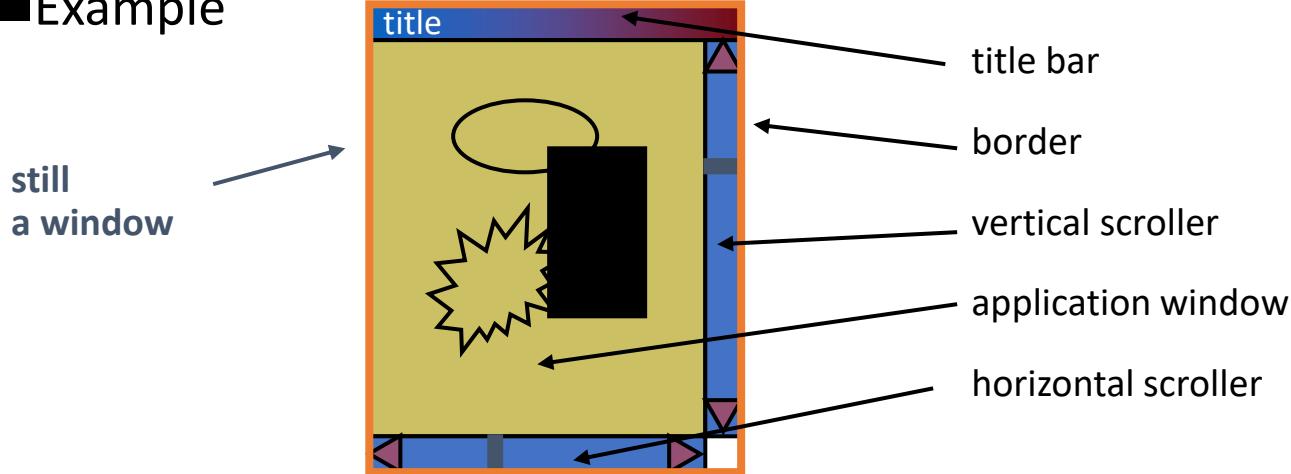
# Structural Patterns

## Decorator (1)

### ■ Intent

- Attach responsibilities to an object dynamically
- Avoid explosion of class number due to (multiple) inheritance

### ■ Example

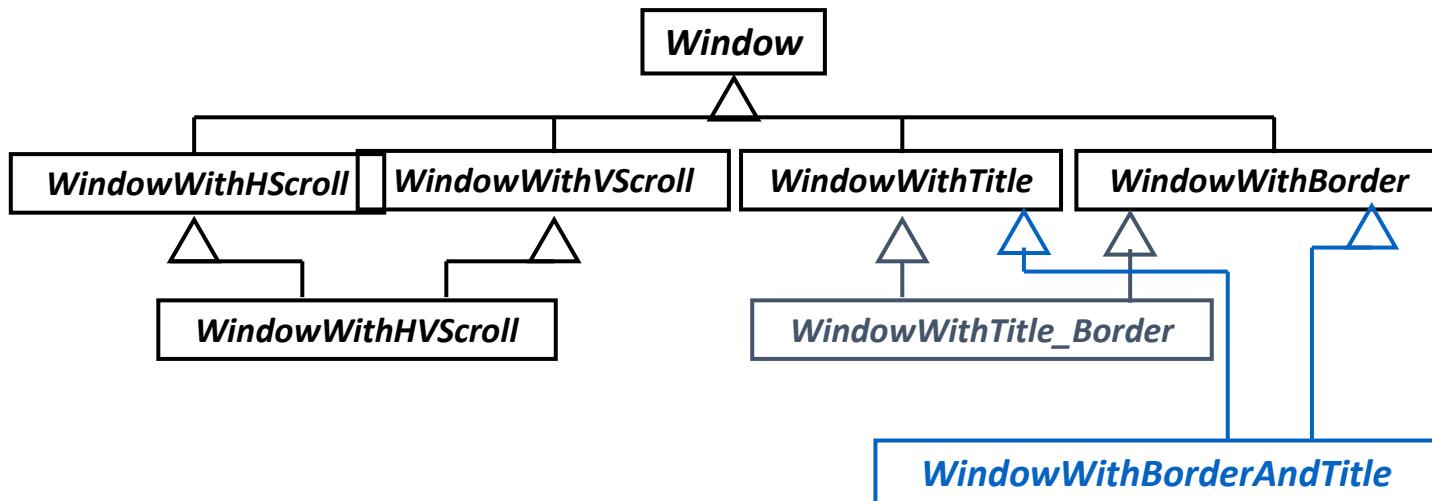


# Structural Patterns

## Decorator (2)

### ■ Example (cont.)

- Using multiple inheritance?



# Structural Patterns

## Decorator (4)

```
class DecoratedWindow extends  
    Window {  
  
    public DecoratedWindow  
        (Window w) {  
        undecorated = w;  
    }  
    void redraw() {  
        undecorated.redraw();  
    }  
  
    private  
        Window undecorated;  
};
```

```
Window w = ...;  
  
Window hsw = new  
    WindowWithHscroll(w);  
  
Window thsw = new  
    WindowWithTitle(hsw);  
  
Window bthsw = new  
    WindowWithBorder(thsw);
```

# Structural Patterns

## Decorator (6)

- **Decorator** is a major pattern in Java
  - e.g., Java IO system

```
BufferedReader bufRead =  
    new BufferedReader(  
        new InputStreamReader(  
            new FileInputStream(filename))) ;
```

# Structural Patterns

## Decorator (7)

- Consequences
  - Transparent enclosure
    - The decorated component has the same properties as the base component (plus some new ones...)
  - Better flexibility than static inheritance
  - Functionalities can be added incrementally
  - Sometimes too general
    - Decorations can be composed in any order
- The decorated object *is not* the base object
  - The system must not rely on object *identity*
- Difficult to “undecorate” in any order
- Lot of small objects
  - Easy to build, not so easy to understand

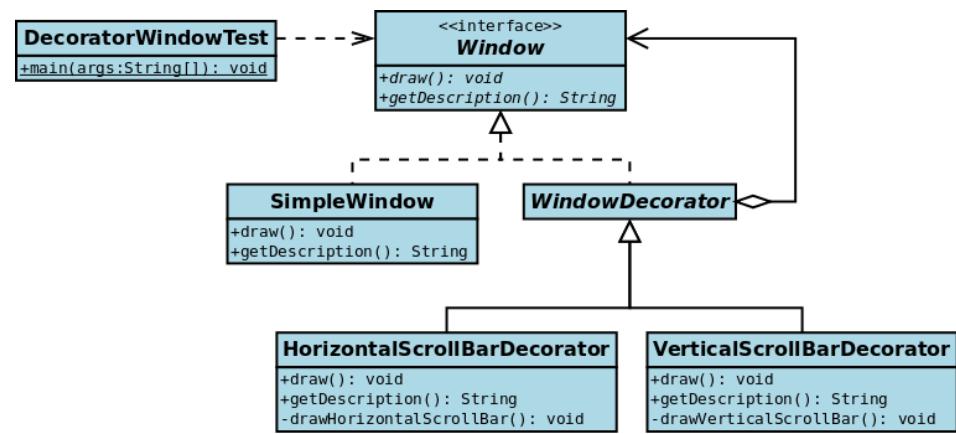
# Decorator

Structural DP

```
// the Window interface
interface Window {
    public void draw(); // draws the Window
    public String getDescription(); // returns a description of the Window
}
```

```
// implementation of a simple Window without any scrollbars
class SimpleWindow implements Window {
    public void draw() {
        // draw window
    }

    public String getDescription() {
        return "simple window";
    }
}
```



# Decorator

Structural DP

```
// abstract decorator class - note that it implements Window
abstract class WindowDecorator implements Window {
    protected Window decoratedWindow; // the Window being decorated

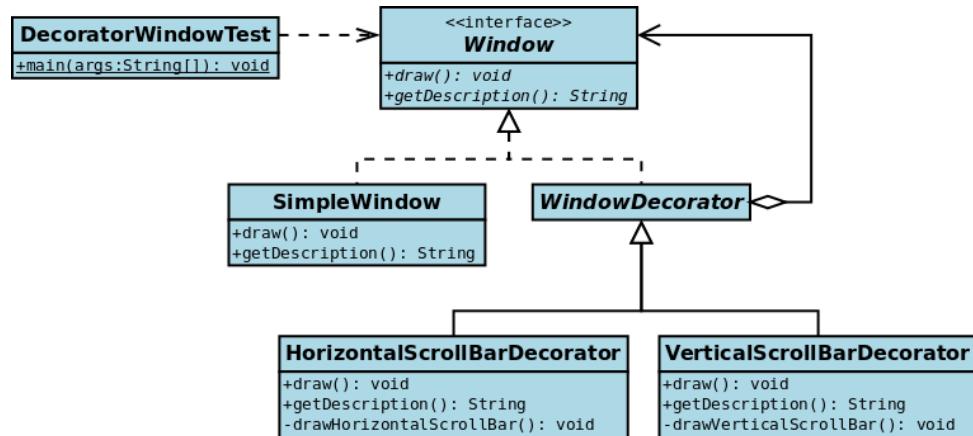
    public WindowDecorator (Window decoratedWindow) {
        this.decoratedWindow = decoratedWindow;
    }
    public void draw() {
        decoratedWindow.draw();
    }
}
```

```
// the first concrete decorator which adds vertical scrollbar functionality
class VerticalScrollBarDecorator extends WindowDecorator {
```

```
    public VerticalScrollBarDecorator (Window decoratedWindow) {
        super(decoratedWindow);
    }
```

```
    public void draw() {
        decoratedWindow.draw();
        drawVerticalScrollBar();
    }
```

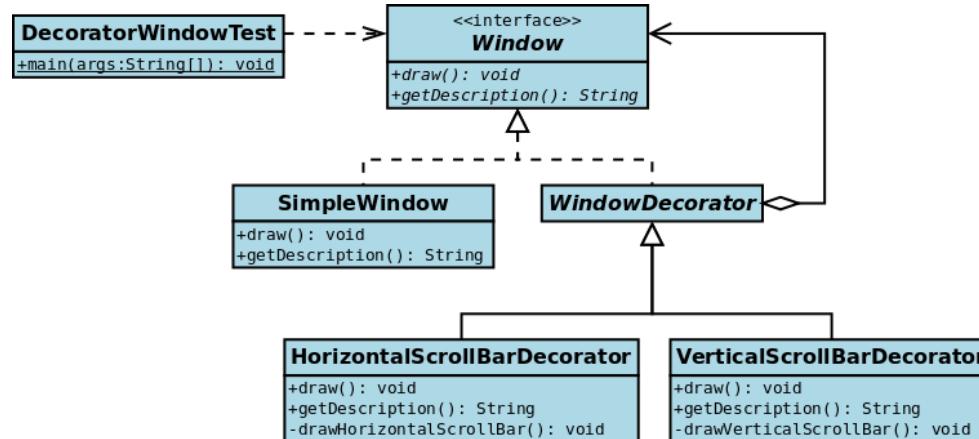
```
    private void drawVerticalScrollBar() {
        // draw the vertical scrollbar
    }
}
```



# Decorator

Structural DP

```
public class DecoratedWindowTest {  
    public static void main(String[] args) {  
        // creating an object to be decorated later  
        Window simpleWindow1 = new SimpleWindow();  
  
        // decorating the simple window with a vertical scroll Bar  
        Window VSB_decoratedWindow = new VerticalScrollBarDecorator(simpleWindow1);  
        VSB_decoratedWindow.draw();  
  
        // further decorating the window with a horizontal Scroll Bar ???  
        Window HSB-VSB-decoratedWindow = new  
            HorizontalScrollBarDecorator(VSB_decoratedWindow1 );  
        HSB-VSB-decoratedWindow.draw();  
    }  
}
```



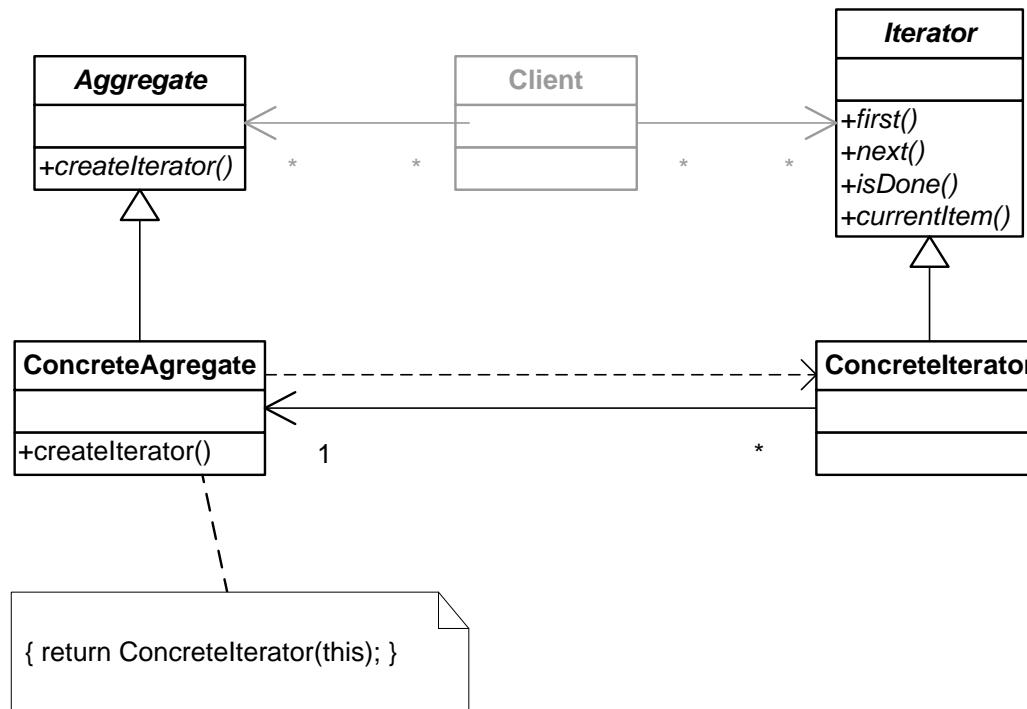
# Outline

- Creational Patterns
  - Singleton
  - Lazy Loading
  - Factory Method
  - Abstract Factory
- Structural Patterns
  - Adapter
  - Decorator
- Behavioural
  - Iterator
  - Mediator
  - Memento
  - Strategy
  - Command
- Architectural
  - MVC
- Anti patterns

# Iterator

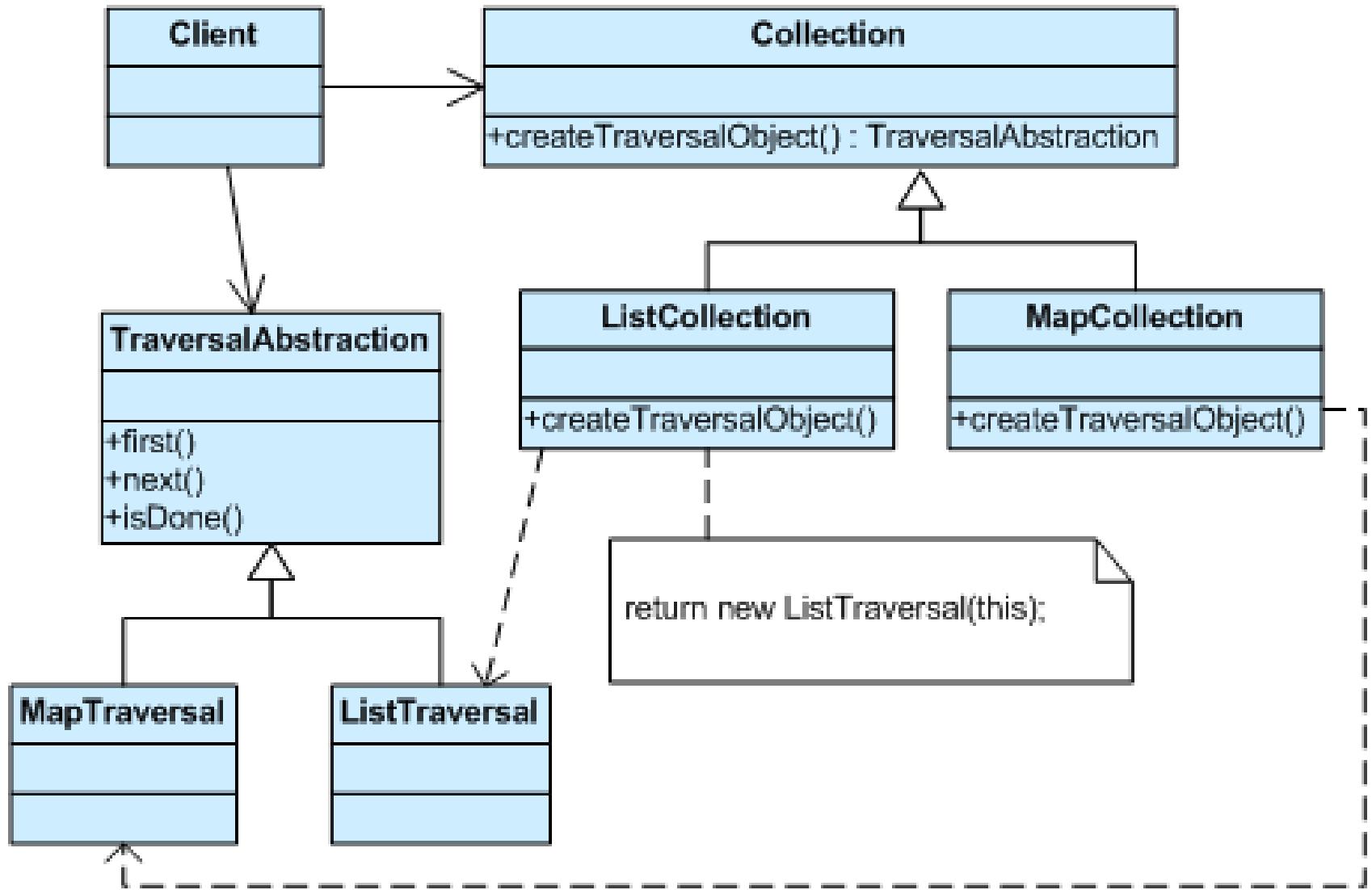
Behavioural DP

- Intent
  - Provide a way to access the elements of an aggregate object (container) sequentially without exposing its underlying representation
- Problem
  - Need to “**abstract**” the **traversal** of wildly different data structures so that algorithms can be defined that are capable of interfacing with each transparently.



# Iterator

Behavioural DP



# Iterator

- Java Array List Iterator

```
ArrayList al = new ArrayList();
// add elements to the array list
al.add("C");
al.add("A");
al.add("E");
al.add("B");
al.add("D");
al.add("F");

// use iterator to display contents of al
System.out.print("Original contents of al: ");
Iterator itr = al.iterator();
while(itr.hasNext()) {

    Object element = itr.next();
    System.out.print(element + " ");

}
```

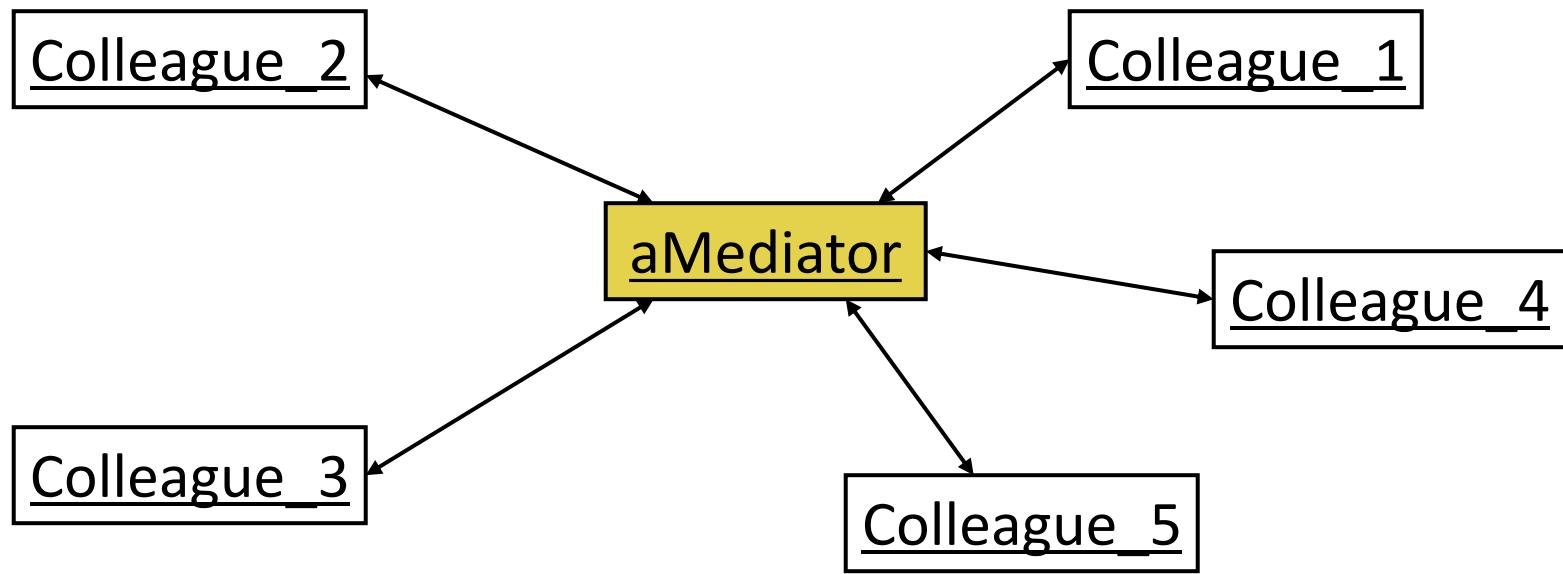
# Mediator

Behavioural DP

- Intent
  - Define an object that encapsulates how a set of objects interact.
  - Mediator promotes ???
    - *loose coupling* by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
- Problem
  - We want to design **reusable components**, but **dependencies** between the potentially reusable pieces demonstrates the “**spaghetti code**” phenomenon

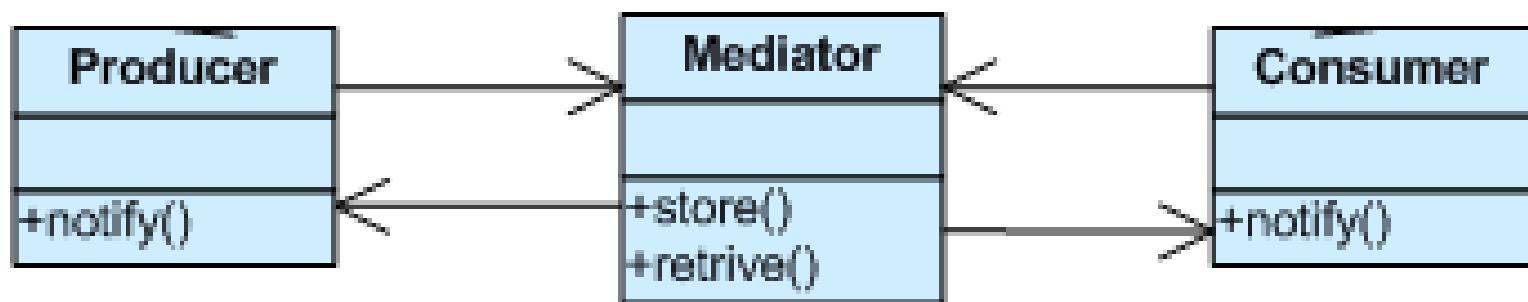
ATC Mediator





# Mediator

Behavioural DP



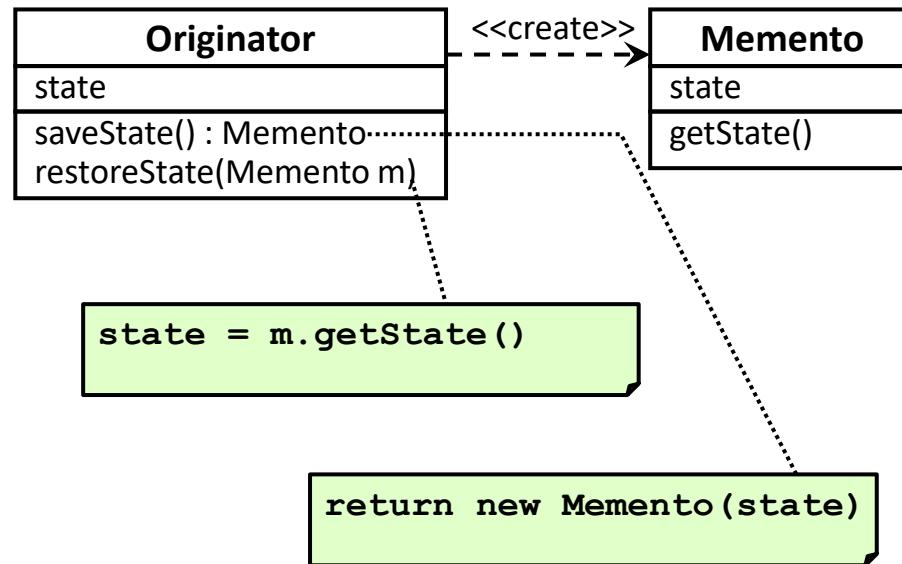
# Behavioral Patterns

## Memento

(1)

### Intent

- Record the state of an object so that the object can be restored to this state later
- Do so without violating the encapsulation

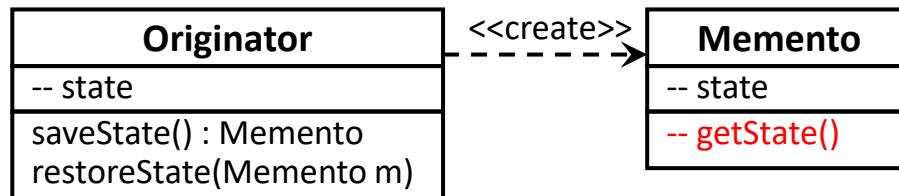


# Behavioral Patterns

## Memento

(2)

- Avoid violating the encapsulation
- A Memento has 2 interfaces
  - A *wide interface*, accessible only from the Originator (access to memento internal data)
  - A *narrow interface*, used by client applications (only pass Memento's as parameters)

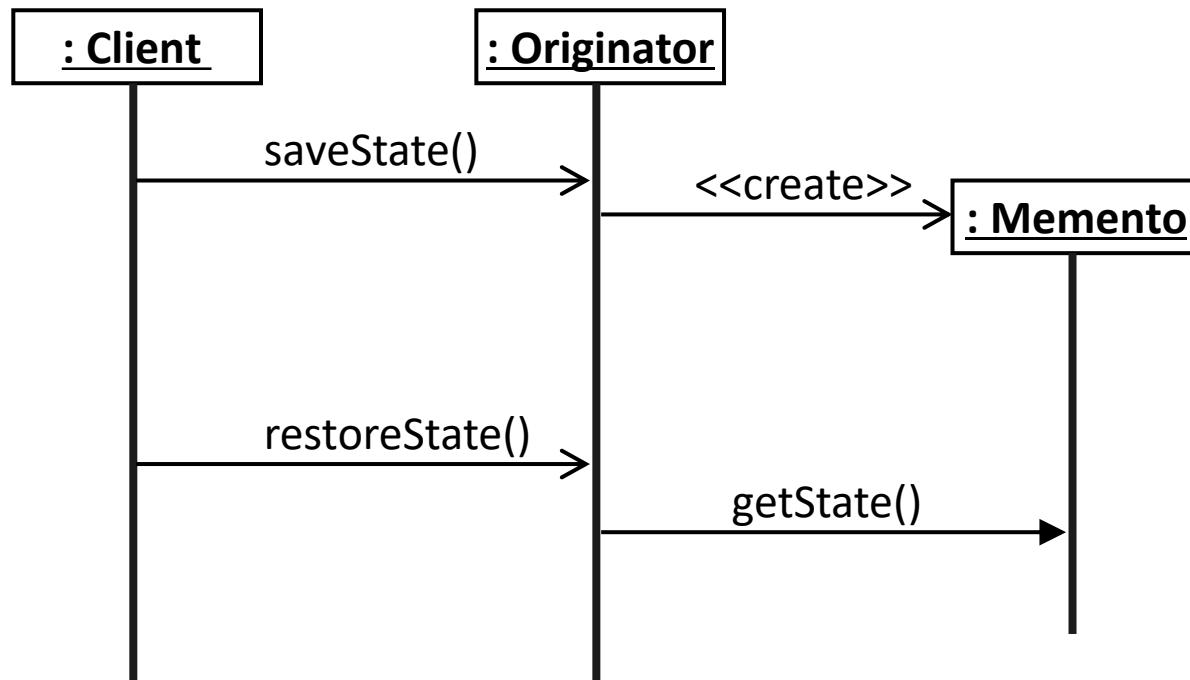


- The Memento are passive objects
- The client application never operates on or consult the Memento data

# Behavioral Patterns

## Memento

(3)



# Behavioral Patterns

## Memento

(4)

```
class State { ... }

class Originator {
    public Memento saveState() {
        return new Memento(_state);
    }

    public void
        restoreState(Memento m)
    {
        _state = m.getState();
    }

    ...
}

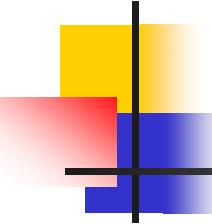
private State _state;
}
```

```
public class Memento {
    // Narrow interface (public)
    // Nothing?
    // Possibly finalize()?

    // Wide interface
    // (package access)
    Memento(State state) {
        _state = state;
    }

    State getState() {
        return _state
    }

    private State _state;
}
```



# Behavioral Patterns

## Memento

(5)

- Consequences
  - Preserve encapsulation
  - Simplify the Originator
    - No need for special memorization operation/data within the Originator itself
  - Memento may be expensive
    - Size of the state information: cost for storing and copying it
  - Some programming languages do not facilitate the two interfaces definition

# Strategy

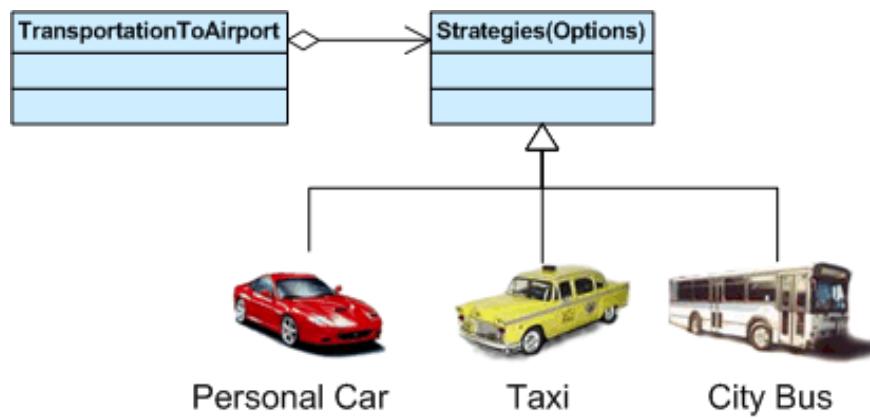
Behavioural DP

## Intent

- Define a **family of algorithms**, encapsulate each one, and make them **interchangeable** to let clients and algorithms vary independently

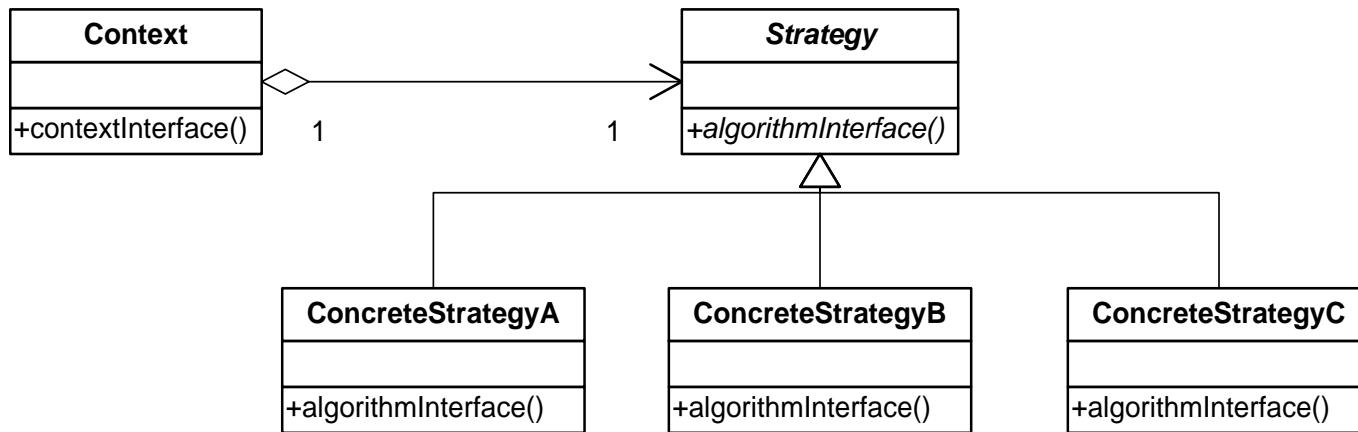
## Applicability

- When an object should be configurable with one of several algorithms *and*:
  - All algorithms can be encapsulated
  - One interface covers all encapsulations



# Strategy

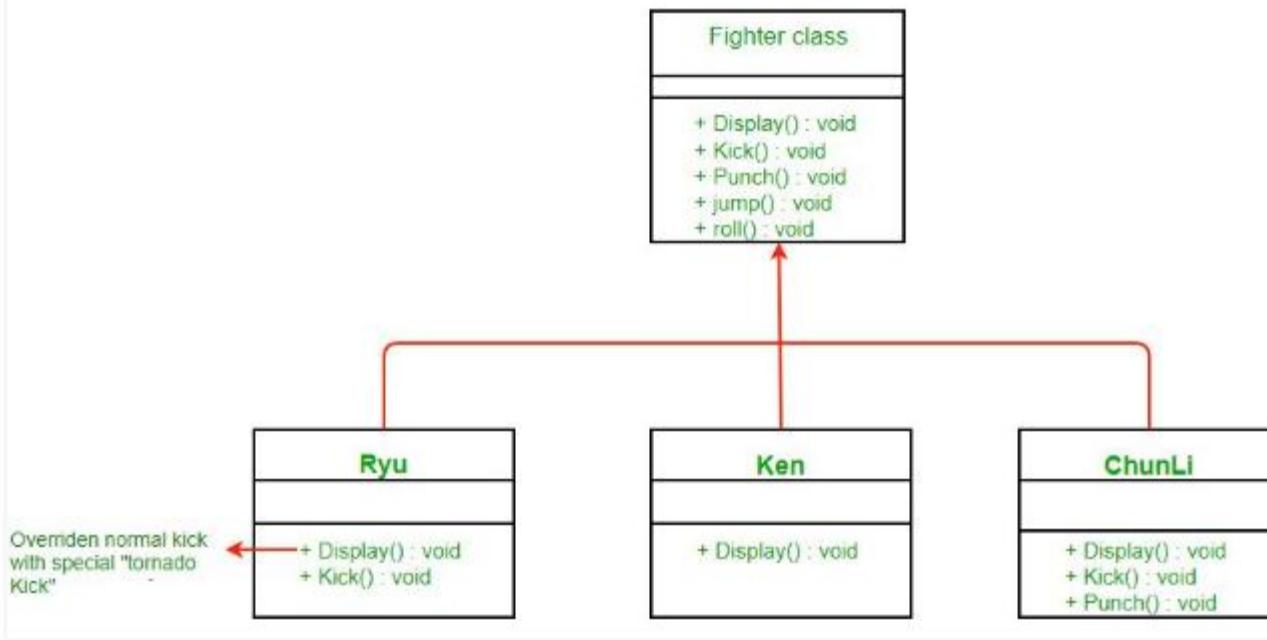
Behavioural DP

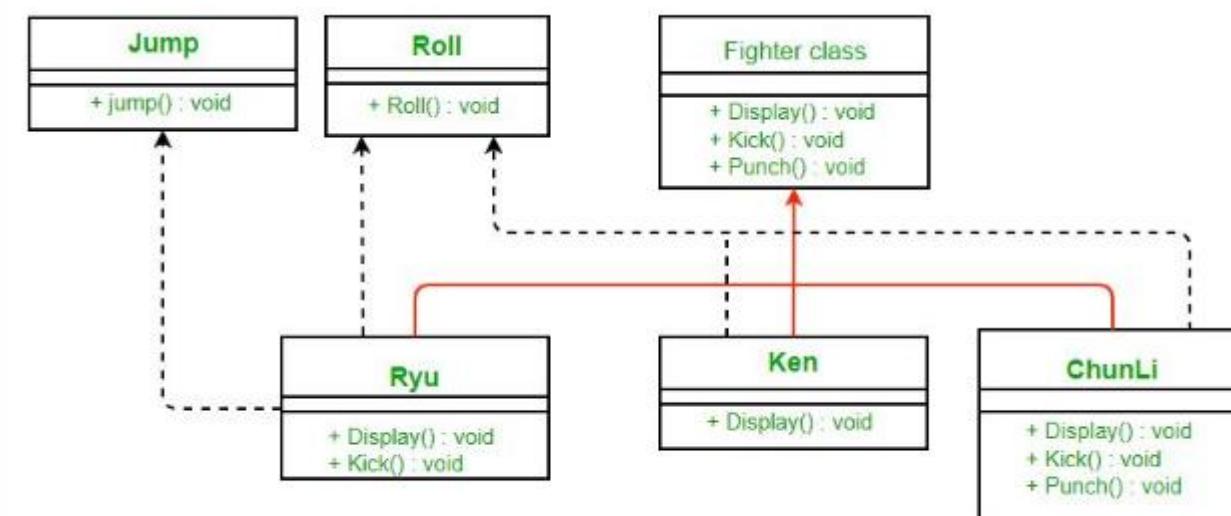


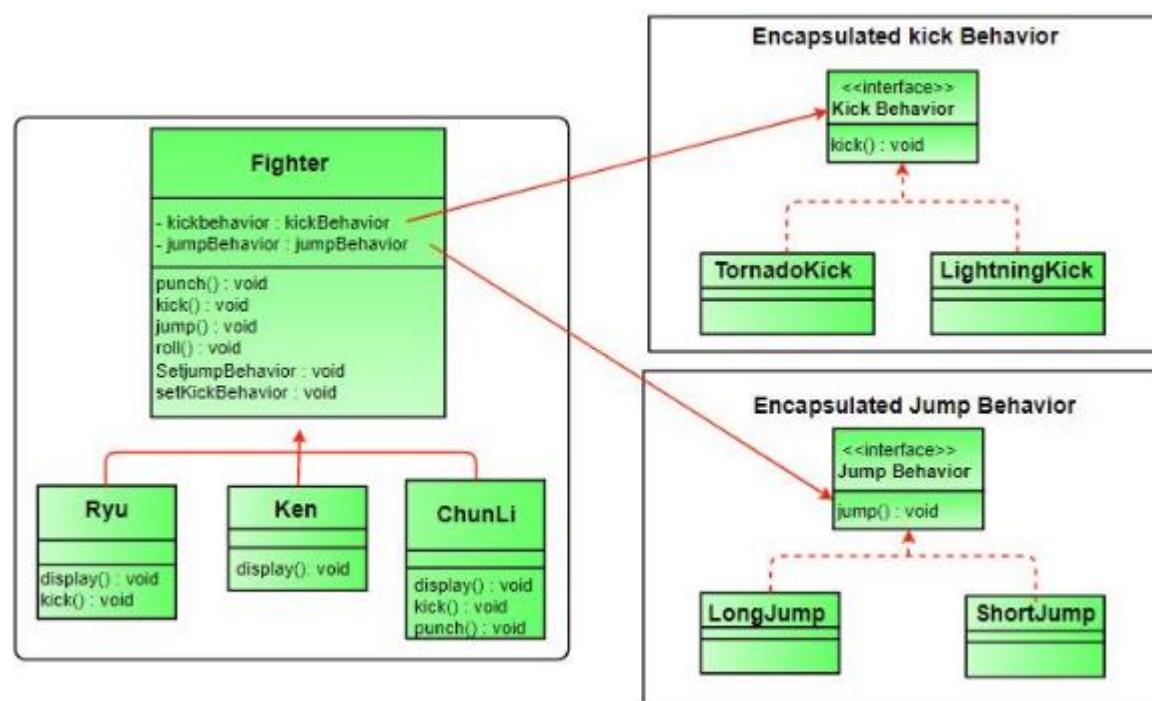
## Consequences

- + Greater flexibility, reuse
- + Can change algorithms dynamically
- Strategy creation & communication overhead
- Inflexible strategy interface

# Strategy Pattern motivation







# Command Pattern

## Command Pattern

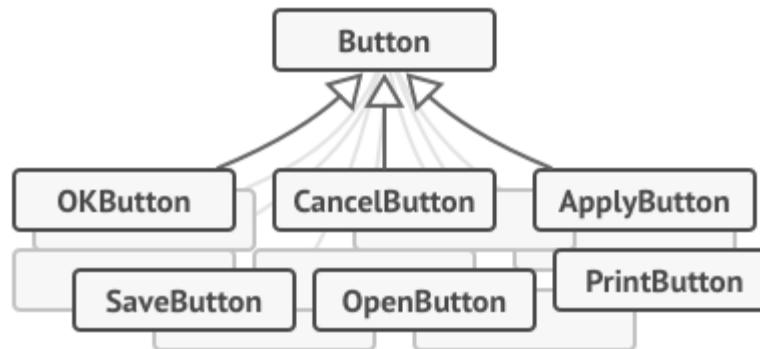
Like [previous](#) articles, let us take up a design problem to understand command pattern. Suppose you are building a home automation system. There is a programmable remote which can be used to turn on and off various items in your home like lights, stereo, AC etc. It looks something like this.

You can do it with simple if-else statements like

```
if (buttonPressed == button1) lights.on()
```

But we need to keep in mind that turning on some devices like stereo comprises of many steps like setting cd, volume etc. Also we can reassign a button to do something else. By using simple if-else we are coding to implementation rather than interface. Also there is tight coupling.

So what we want to achieve is a design that provides loose coupling and remote control should not have much information about a particular device. The command pattern helps us do that.

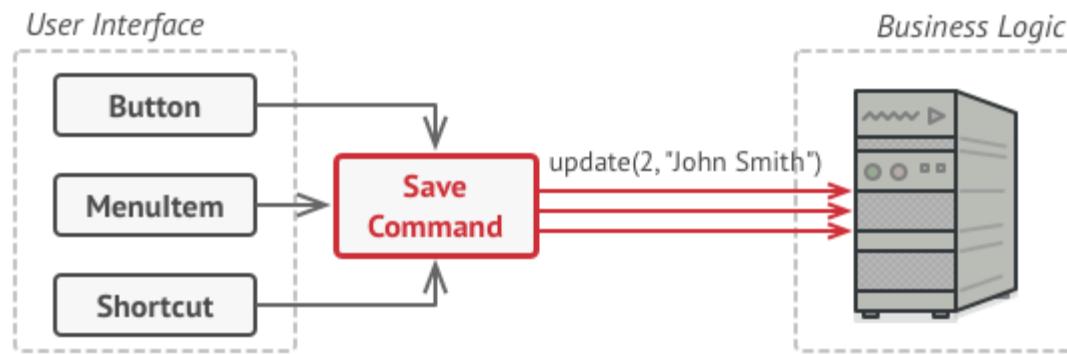


*Lots of button subclasses. What can go wrong?*

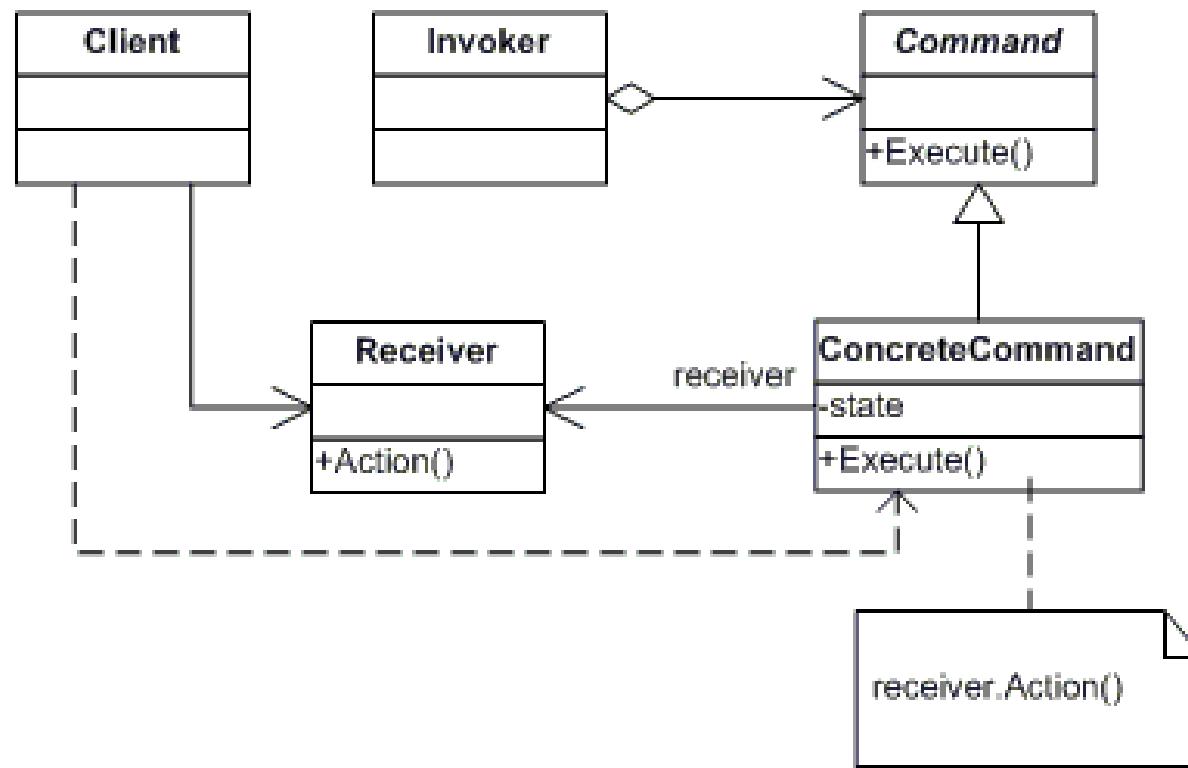


*Several classes implement the same functionality.*

- The Command pattern suggests that GUI objects shouldn't send these requests directly. Instead, you should extract all of the request details, such as the object being called, the name of the method and the list of arguments into a separate *command* class with a single method that triggers this request.
- Command objects serve as links between various GUI and business logic objects. From now on, the GUI object doesn't need to know what business logic object will receive the request and how it'll be processed. The GUI object just triggers the command, which handles all the details.



*Accessing the business logic layer via a command.*



# Command

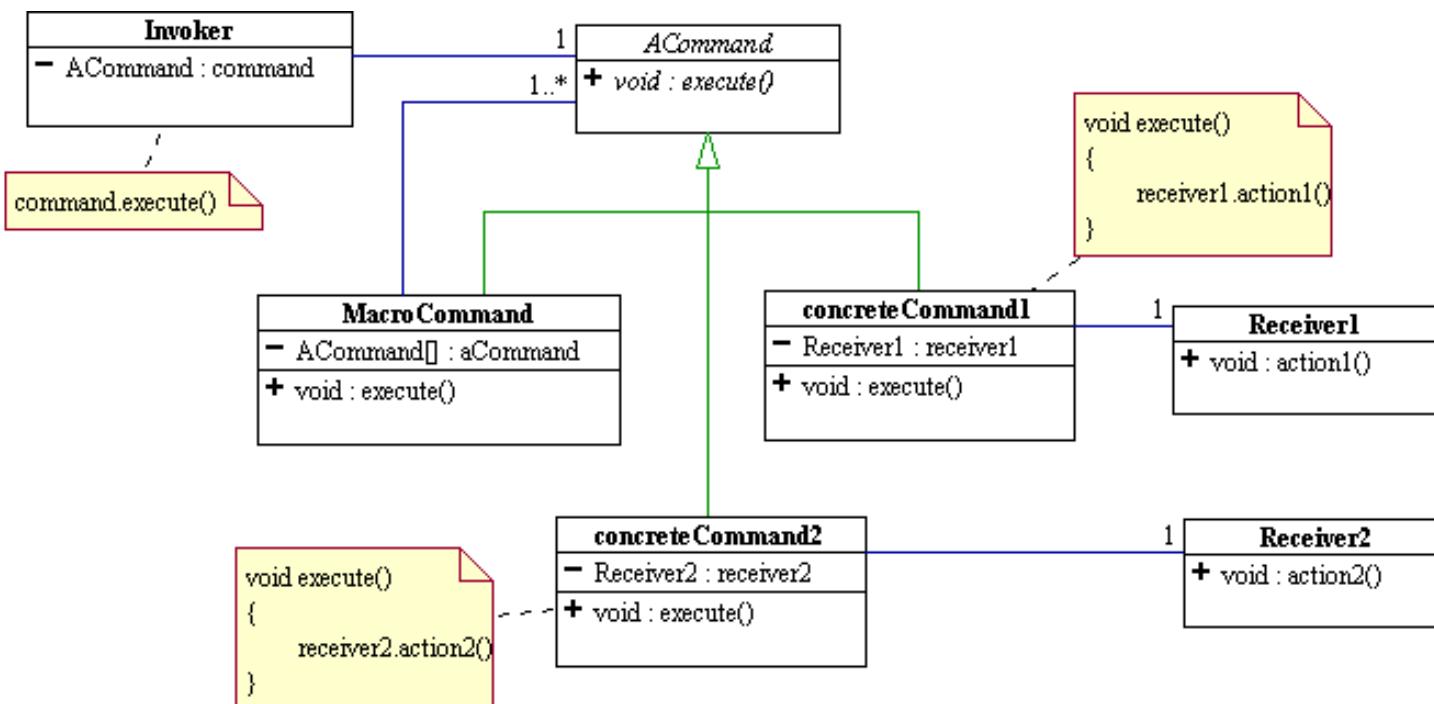
Behavioural DP

- Intent
  - Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- Problem
  - Need to issue requests to objects without knowing anything about the operation being requested or the receiver of the request.
  - When two objects communicate, often one object is sending a command to the other object to perform a particular function
    - The first object (the "invoker") holds a reference to the second (the "recipient")
    - The invoker executes a specific method on the recipient to send the command.
  - *What if the invoker is not aware of, or does not care who the recipient is?*
    - The invoker simply wants to abstractly issue the command
      - E.g. Program Menu

# Command

Behavioural DP

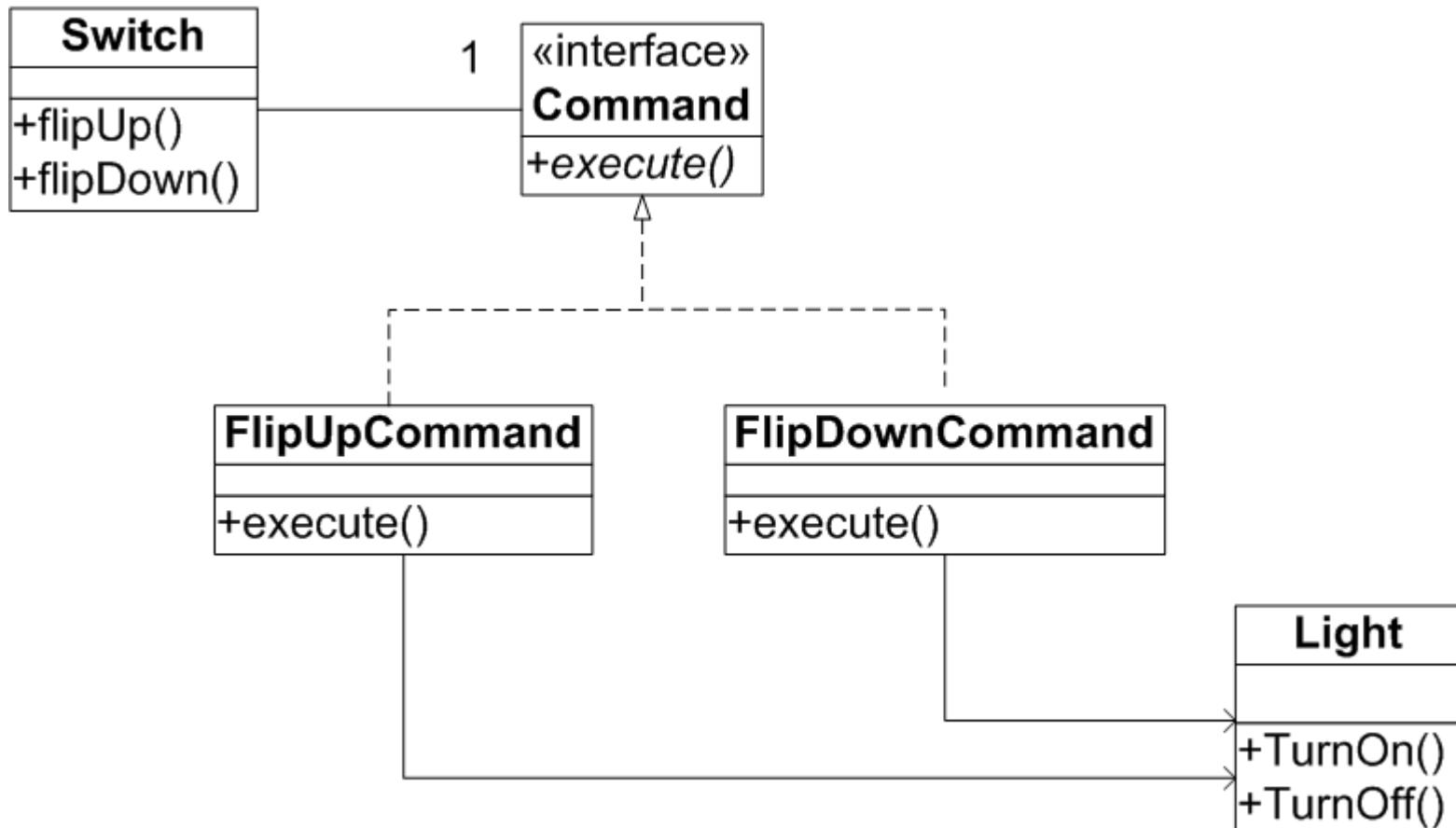
- The Command design pattern encapsulates the concept of the command into an object. The invoker holds a reference to the command object rather than to the recipient.
- The invoker **sends the command to the command object** by executing a specific method on it. The command object is then responsible for dispatching the command to a specific recipient to get the job done.



# Command

Behavioural DP

Design a switching system for turning on and off a light. Make the switch as reusable as possible (the same switch can be used for turning on and off an engine)



# Command

Behavioural DP

Design a switching system for turning on and off a light. Make the switch as reusable as possible (the same switch can be used for turning on and off an engine)

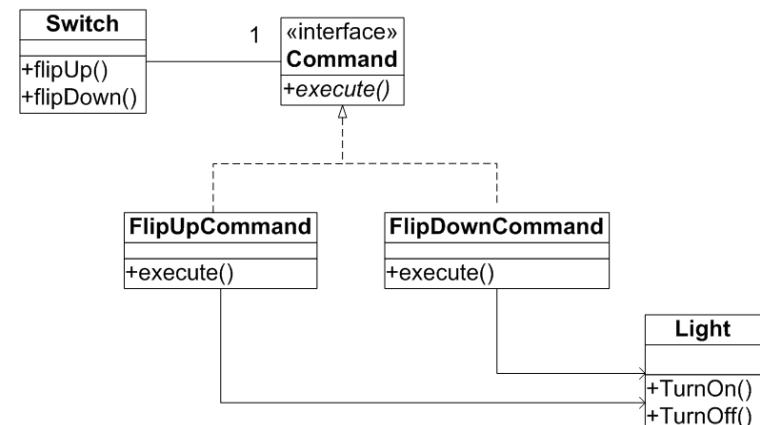
```
/* The Invoker class */
public class Switch {

    private Command flipUpCommand;
    private Command flipDownCommand;

    public Switch(Command flipUpCmd, Command flipDownCmd) {
        this.flipUpCommand = flipUpCmd;
        this.flipDownCommand = flipDownCmd;
    }

    public void flipUp() {
        flipUpCommand.execute();
    }

    public void flipDown() {
        flipDownCommand.execute();
    }
}
```



# Command

Behavioural DP

```
/* The Command interface */
public interface Command {
    void execute();
}

/* The Command for turning the light on */
public class FlipUpCommand implements Command {
    private Light theLight;

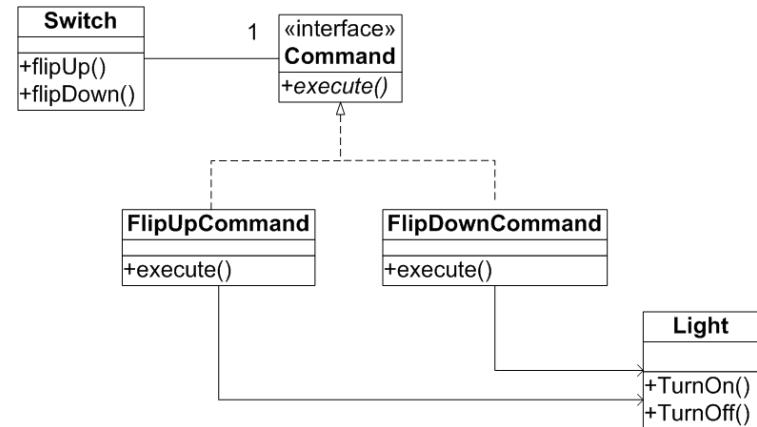
    public FlipUpCommand(Light light) {
        this.theLight=light;
    }

    public void execute(){
        theLight.turnOn();
    }
}

/* The Command for turning the light off */
public class FlipDownCommand implements Command {
    private Light theLight;

    public FlipDownCommand(Light light) {
        this.theLight=light;
    }

    public void execute() {
        theLight.turnOff();
    }
}
```



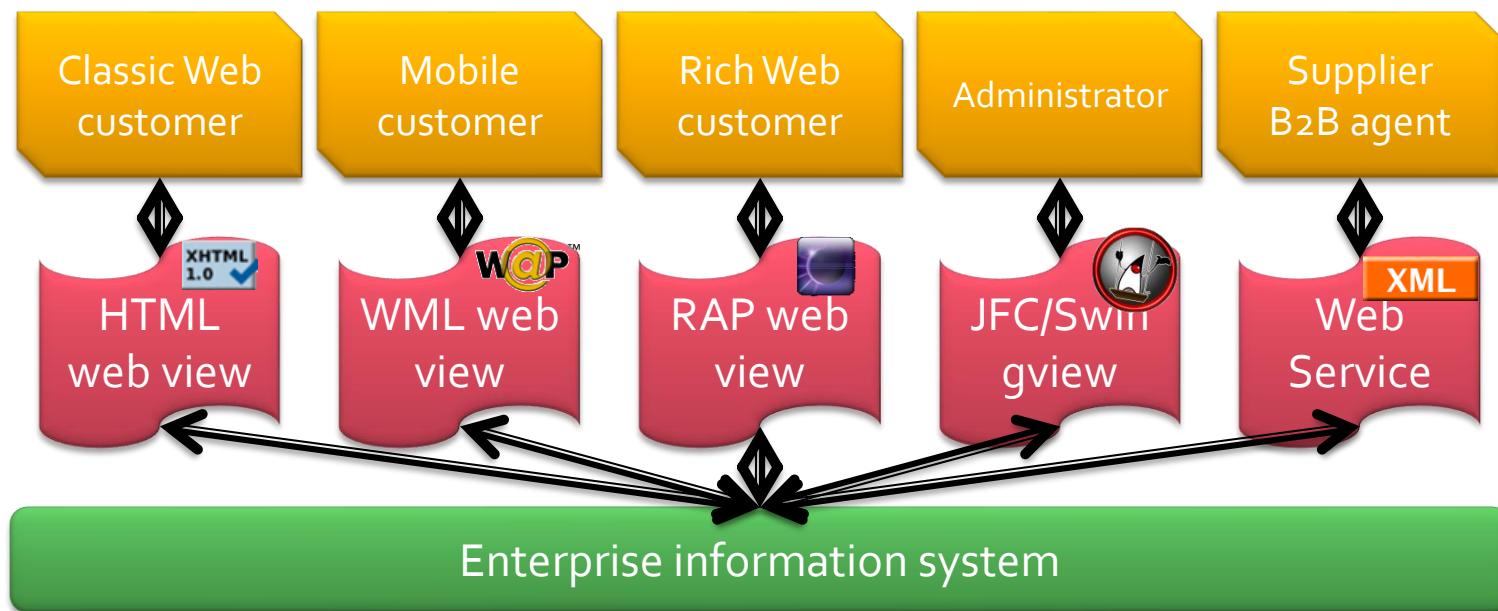
# Outline

- Creational Patterns
  - Singleton
  - Lazy Loading
  - Factory Method
  - Abstract Factory
- Structural Patterns
  - Adapter
  - Decorator
- Behavioural
  - Iterator
  - Mediator
  - Memento
  - Strategy
  - Command
- Architectural
  - MVC
- Anti patterns

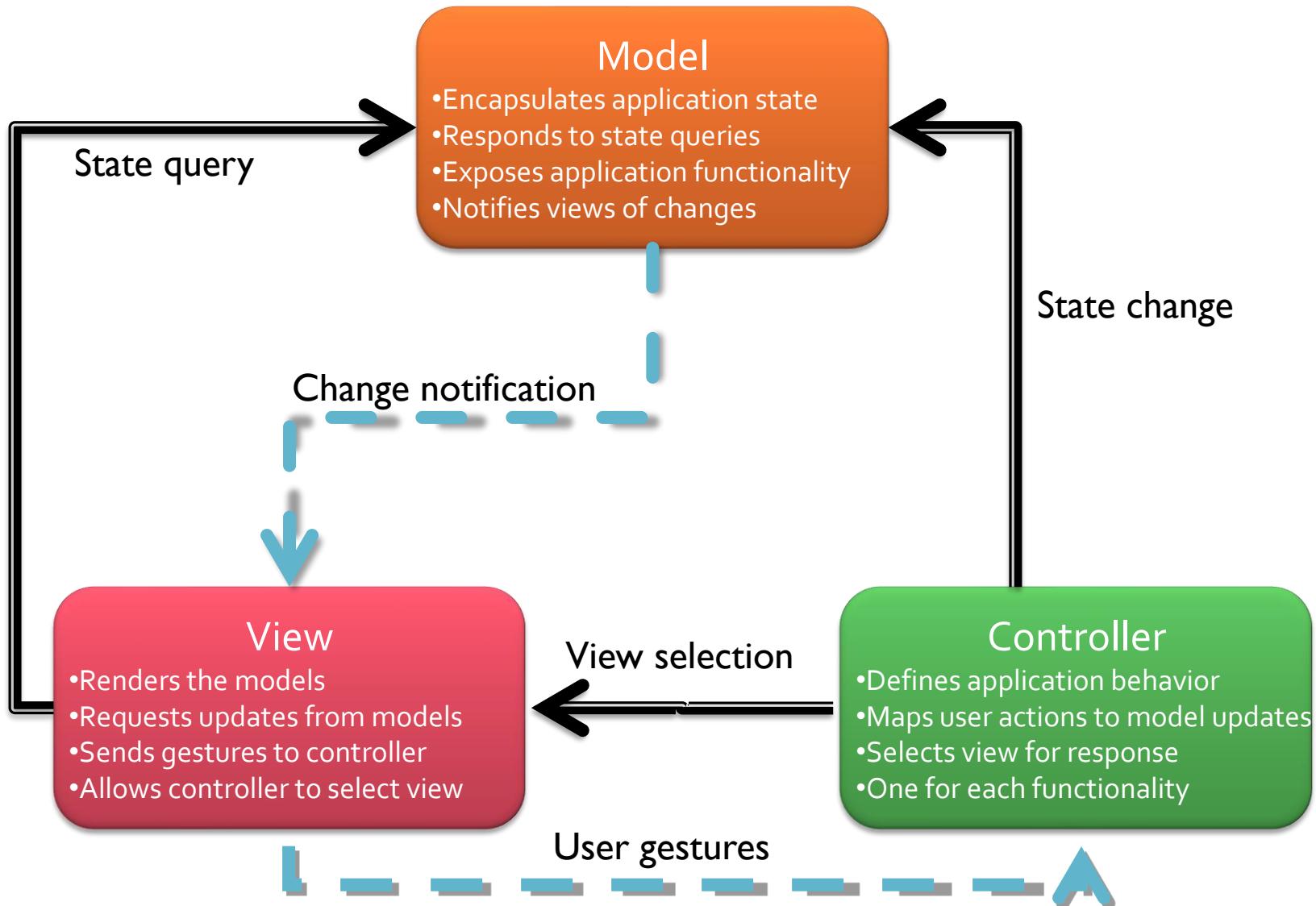
# Model-View-Controller (MVC)

Architectural DP

- Application presents content to users in numerous ways containing various data.



# MVC



# Outline

- Creational Patterns
  - Singleton
  - Lazy Loading
  - Factory Method
  - Abstract Factory
- Structural Patterns
  - Adapter
  - Decorator
- Behavioural
  - Iterator
  - Mediator
  - Memento
  - Strategy
  - Command
- Architectural
  - MVC
- Anti patterns

# AntiPatterns

- An industry vocabulary for **the common defective** processes and implementations within organizations.
  - A higher-level **vocabulary simplifies communication** between software practitioners and enables concise description of higher-level concepts.
- The AntiPattern may be the result of a manager or developer:
  - Not knowing any better
  - Not having sufficient knowledge or experience in solving a particular type of problem
  - Having applied a perfectly good pattern in the wrong context.

# AntiPatterns



## ■ Software Development AntiPatterns

- Describe useful forms of software refactoring
- A concept similar to *Code Smell*



## ■ Software Architecture AntiPatterns

- focus on the system-level and enterprise-level structure of applications and components.



## ■ Software Project Management AntiPatterns

- Identify some of the key scenarios in which human-related issues are destructive to software processes.

# Sources

- <http://sourcemaking.com/>
- [http://en.wikipedia.org/wiki/Design\\_pattern\\_\(computer\\_science\)](http://en.wikipedia.org/wiki/Design_pattern_(computer_science))
- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Gang of Four (GoF)) Elements of Reusable Object-Oriented Software

# Architectural styles

# Agenda

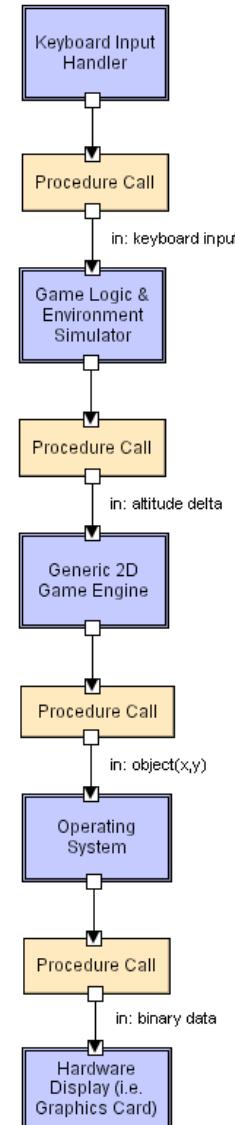
- Types of architectural styles
- Basic decomposition techniques
  - layering, tiering
- Architectural styles
  - Pipes and filters
  - Repository
  - Client/Server
    - two-tiers; three-tiers; n-tiers
  - Model/View/Controller
  - Service-Oriented
  - Peer-To-Peer

# Layers

- An architecture which has a hierarchical structure, consisting of an ordered set of layers, is *layered*
- A layer is a set of subsystems which are able to provide related services, that can be realized by exploiting services from other layers
- A layer depends only from its lower layers (i.e., layers which are located at a lower level into the architecture)
  - A layer is only aware of lower layers

# Layers: component diagram

- Connectors for layered systems are often procedure calls
- Each level implements a different VM with a specific input language

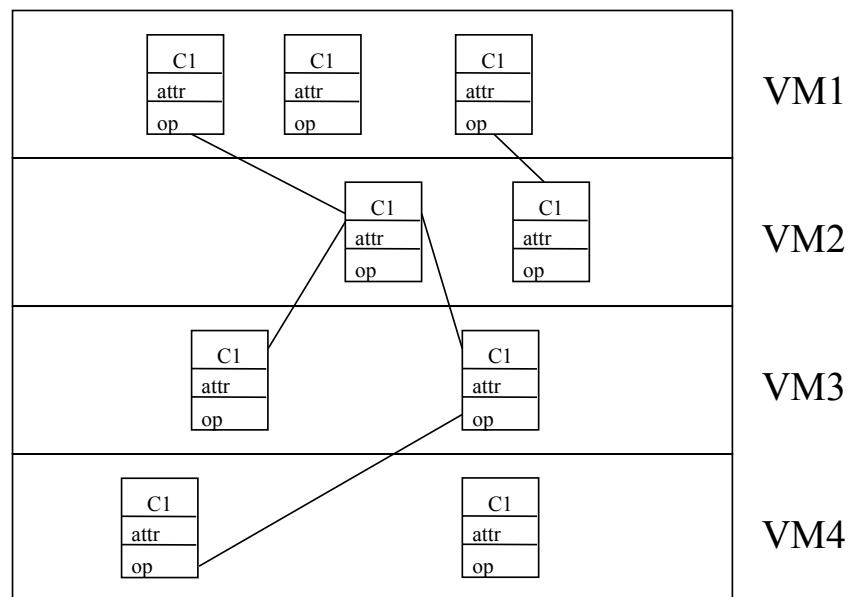


# Closed vs open layers

- **Closed architecture:** the  $i$ -th layer can only have access to the layer  $(i-1)$ -th
- **Open architecture:** the  $i$ -th layer can have access to all the underlying layers (i.e., the layers lower than  $i$ )

# Layers – Closed architecture

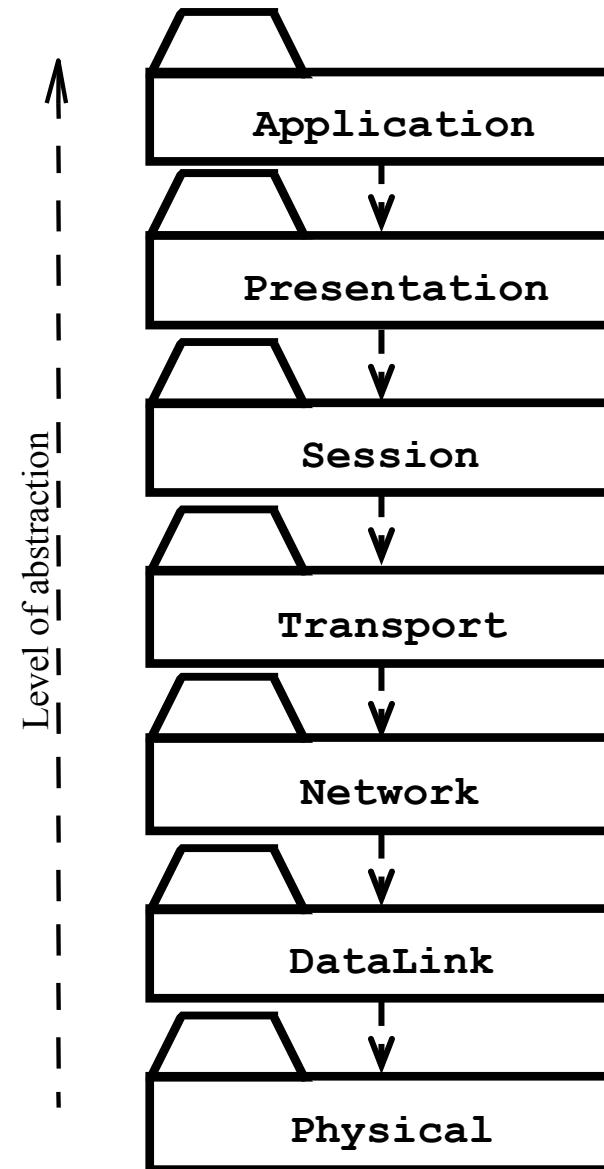
- The i-th layer can only invoke the services and the operations which are provided by the layer (i-1)-th
- The main goals are the system maintainability and high portability (eg. Virtualization: each layer i defines a Virtual Machine - VMi)



# Closed layers: ISO/OSI stack

The ISO/OSI reference model defines 7 network layers, characterized by an increasing level of abstraction

Eg. The Internet Protocol (IP) defines a VM at the network level able to identify net hosts and transmit single packets from host to host

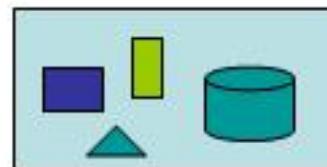


# Tiers

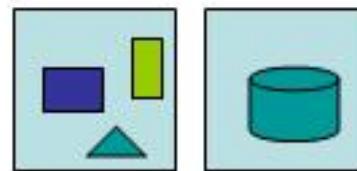
- Another approach to managing complexity consists of *partitioning* a system into sub-systems (*peers*), which are responsible for a class of services

# The Computing Evolution

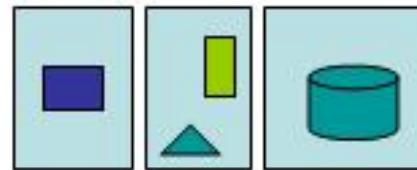
**Monolithic  
(one tier)**



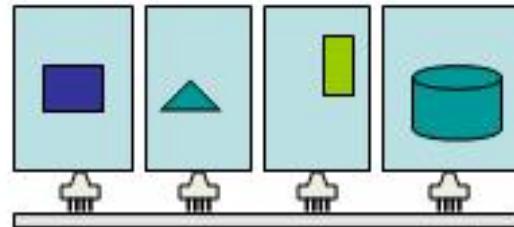
**Client / Server**



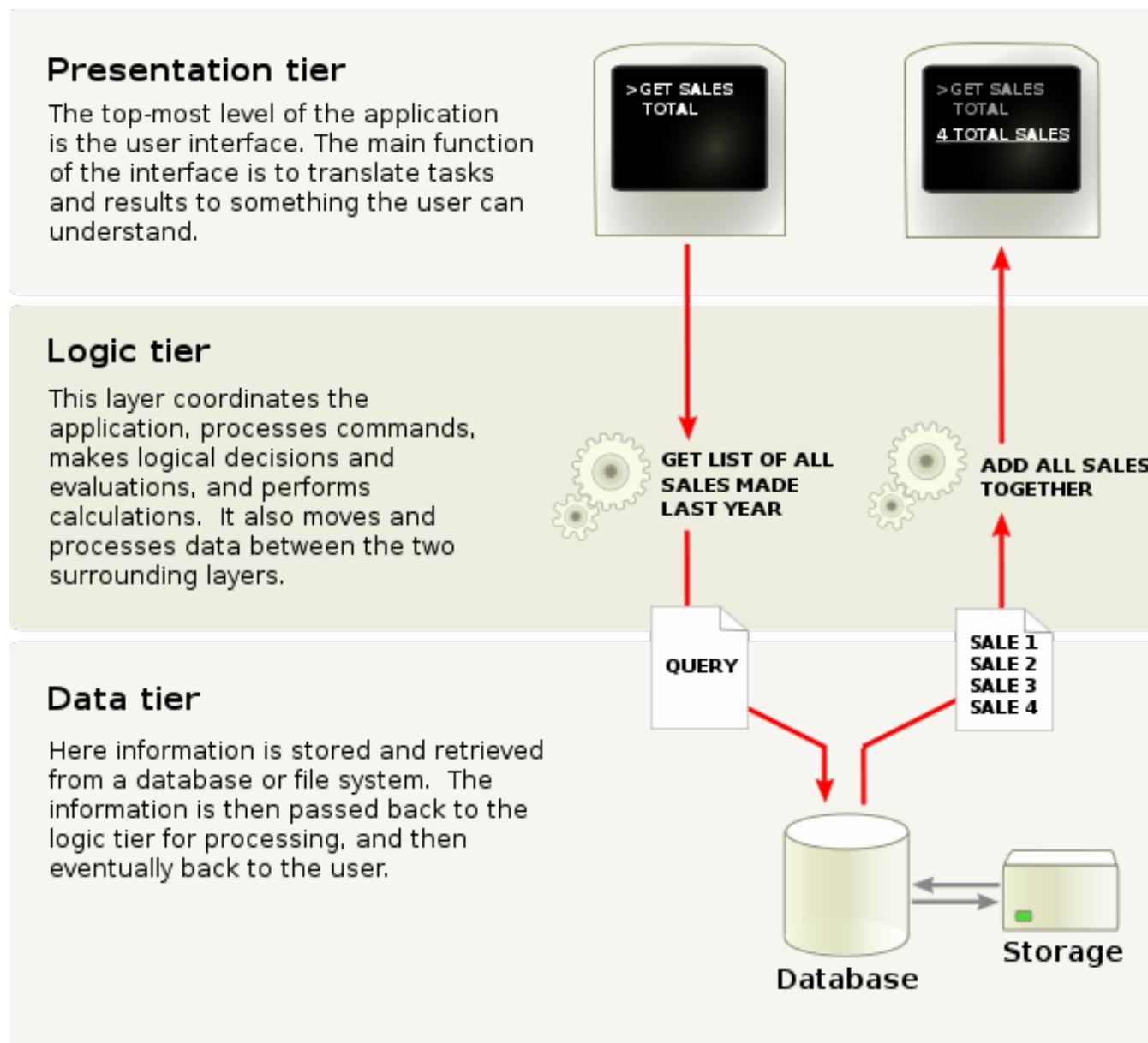
**3-Tier**



**N-Service**



# Typical tiered architecture



# Layers and tiers

- Typically, a complete decomposition of a given system comes from both layering and partitioning
  - First, the system is divided into top level subsystems which are responsible for certain functionalities (*partitioning*)
  - Second, each subsystem is organized into several layers, if necessary, up to the definition of *simple enough* layers

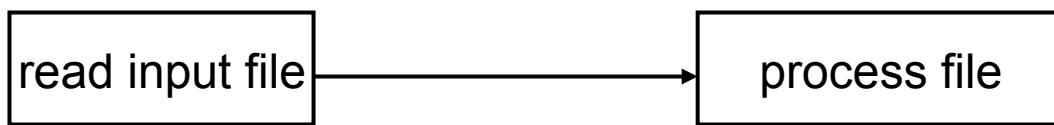
# Architectural styles

- Several architectural styles have been defined in the literature of software engineering.
- They can be used as the basis for configuring software architectures.
- The basic styles include:
  - Pipes and filters
  - Repository
  - Client/Server: two-tiers; three-tiers; n-tiers
  - Model/View/Controller
  - Service-Oriented
  - Peer-To-Peer

# Pipes and Filters style

# Pipe and Filter Architecture

- Main components:
  - Filter: process the a stream of input data to some output data
  - Pipe: a channel that allows the flow of data

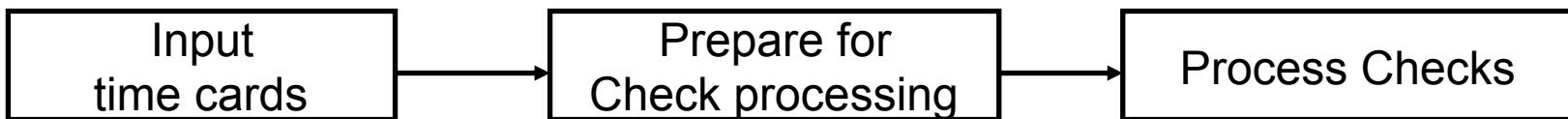


filter  
pipe

This architecture style focuses  
on the dynamic (interaction)  
rather than the structural

# Pipe-Filter example

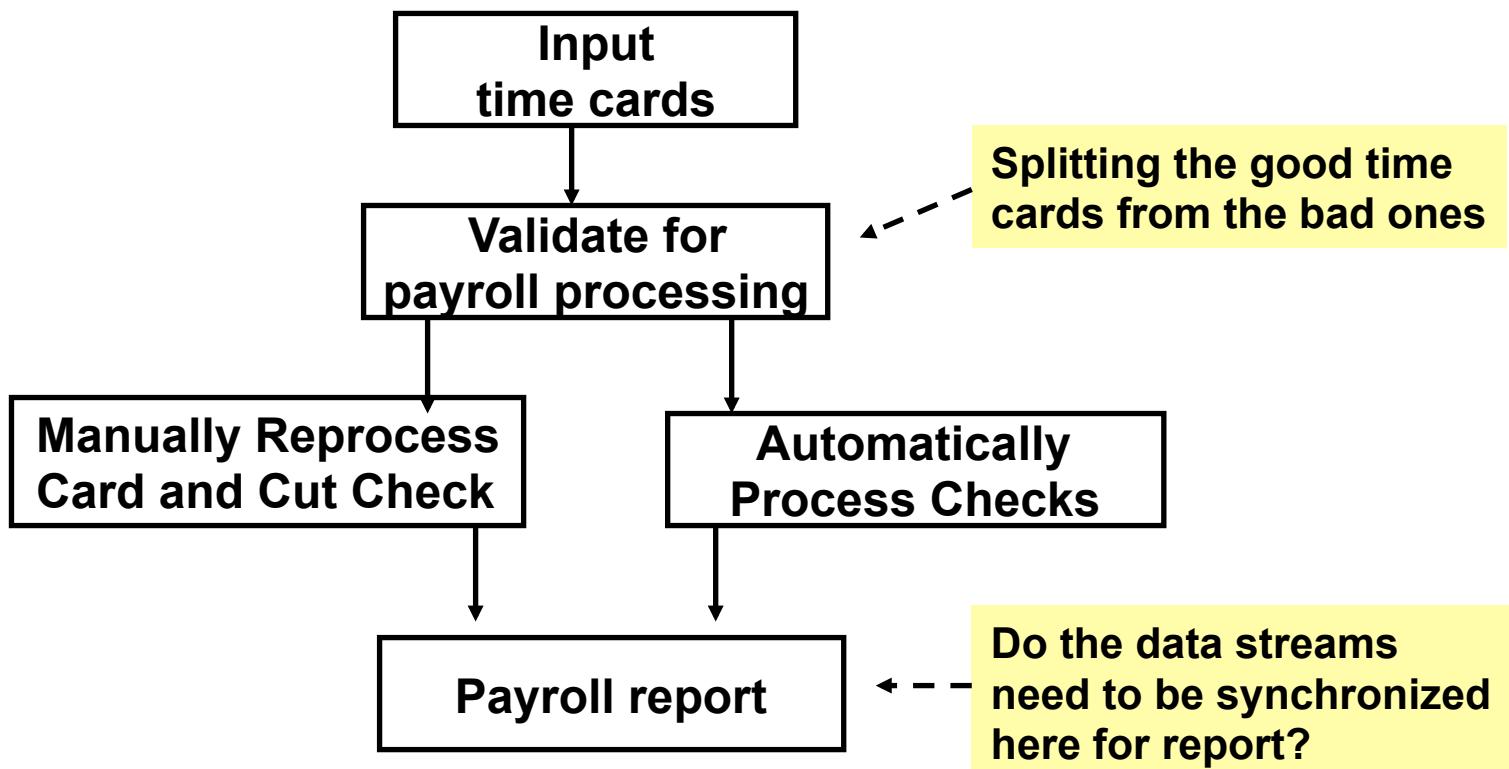
- The high level design solution is decomposed into 2 parts (filters and pipes):
  - *Filter is a service that transforms a stream of input data into a stream of output data*
  - *Pipe is a mechanism or conduit through which the data flows from one filter to another*



Problems that require batch file processing seem to fit this: [payroll](#) and [compilers](#)

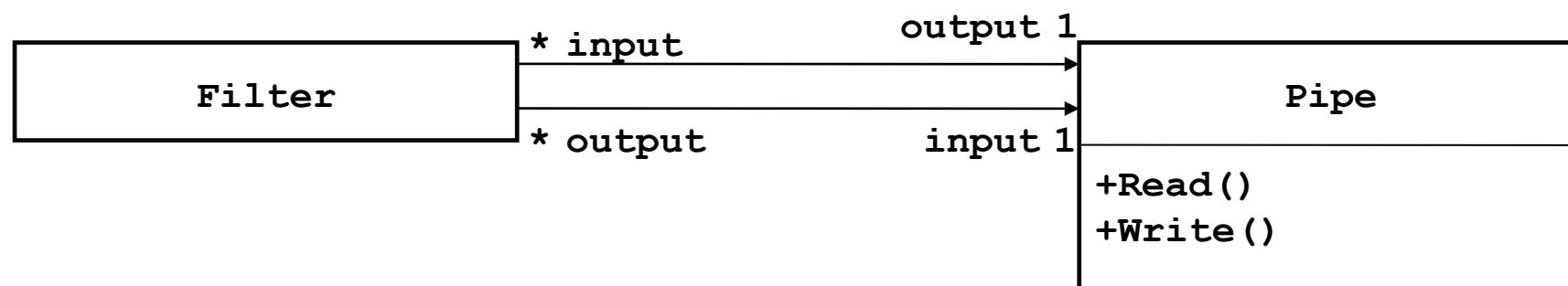
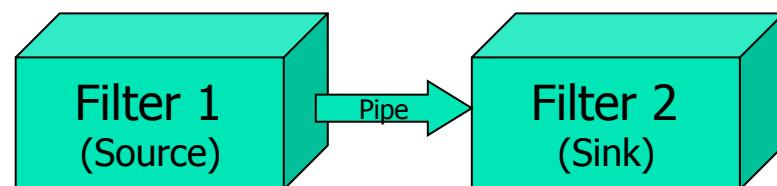
# Pipe – Filter with error processing

- Even if interactive error processing is difficult, batch error processing can be done with a pipe and filter architecture



# Pipes and filters

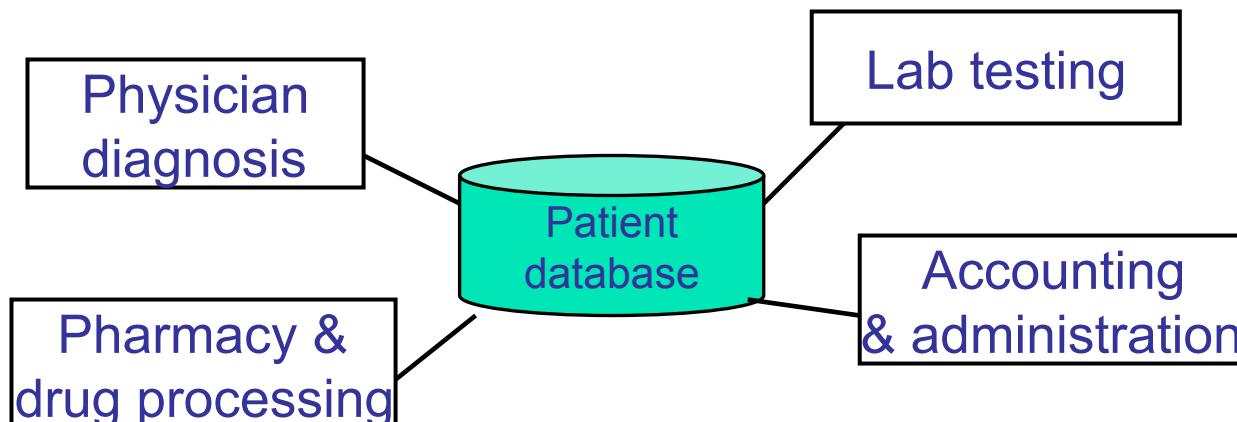
- Pipe: communication channel
- Filter: computing component
  - Filter 1 may only send data to Filter 2
  - Filter 2 may only receive data from Filter 1
  - Filter 1 may not receive data
  - Filter 2 may not send data
  - Pipe is the data transport mechanism



# Repository-based style

# Shared Data

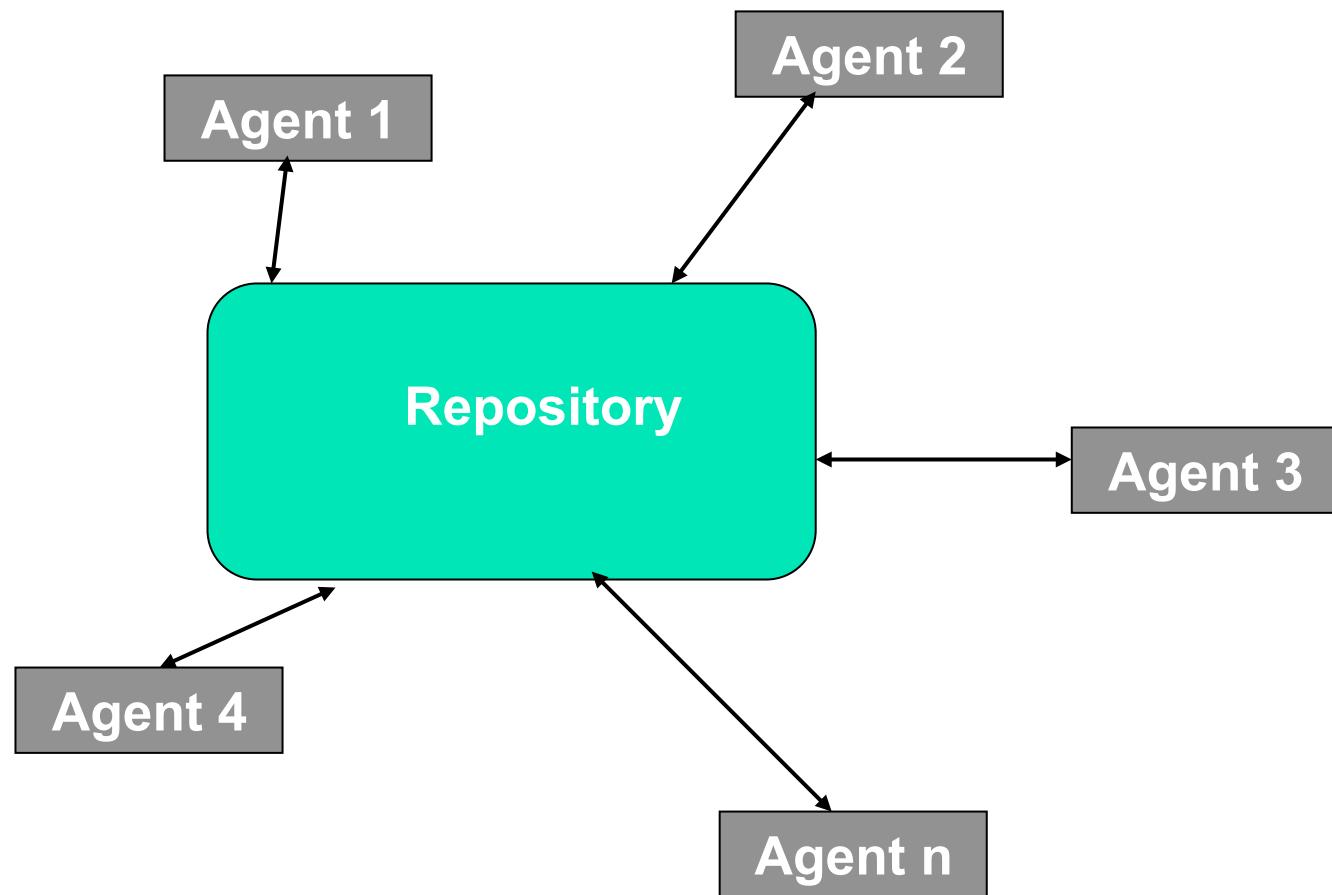
- Very common in information systems where data is shared among different functions
- A repository is a *shared data-store* with two variants:
  - *Repository style*: the participating parties check the data-store for changes
  - *Blackboard (or tuple space) style*: the data-store alerts the participating parties whenever there is a data-store change (trigger)



Problems that fit this style have the following properties:

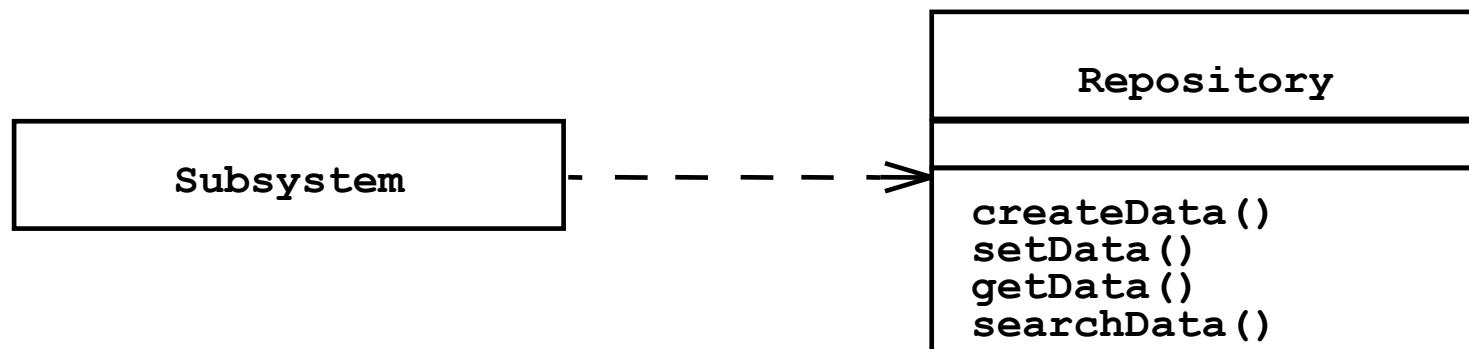
1. All the functionalities work off a shared data-store
2. Any change to the data-store may affect all or some of the functions
3. All the functionalities need the information from the data-store

# Repository Architecture

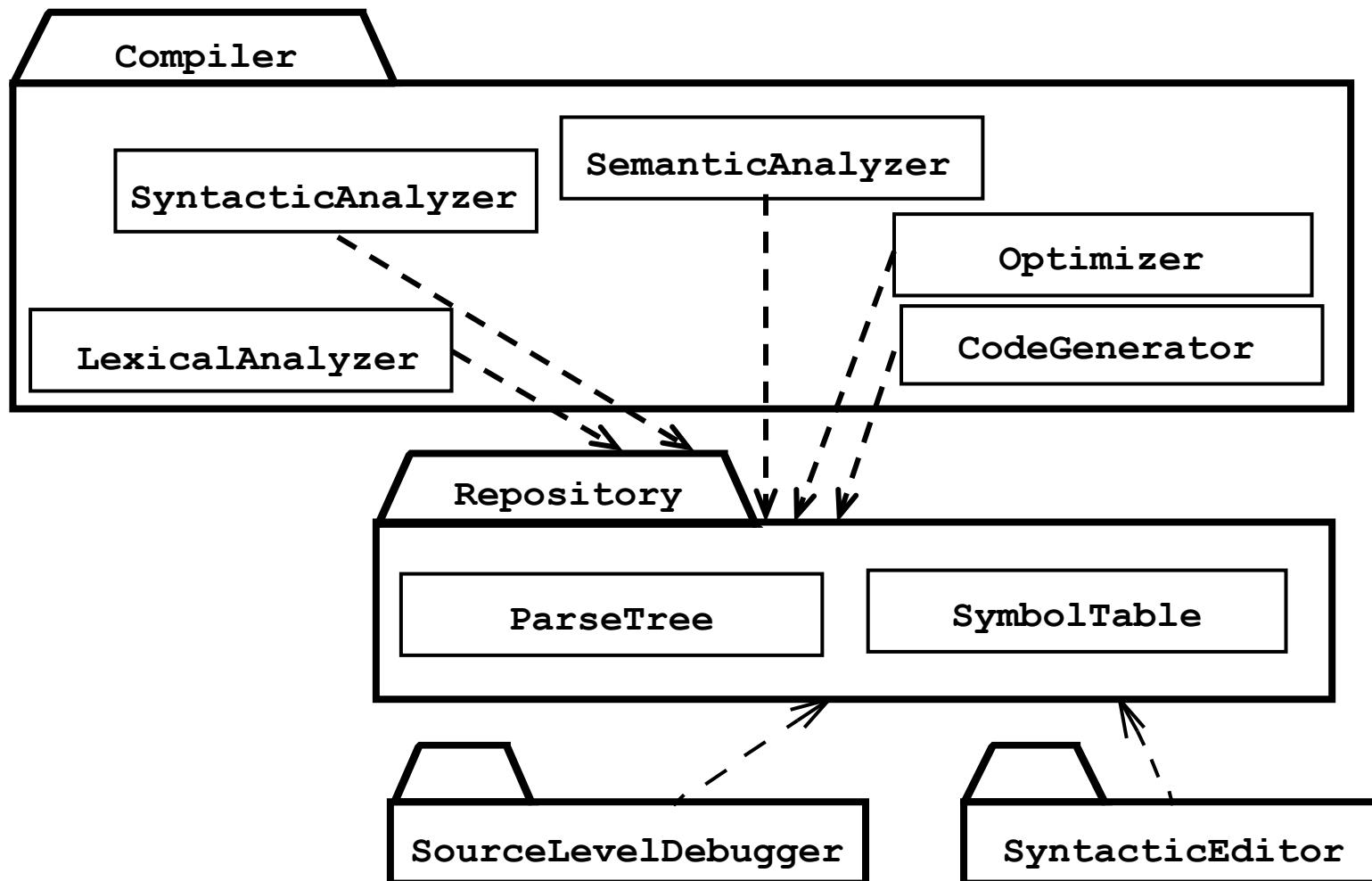


# Repository Architecture

- The subsystems use and modify (i.e., they have access to) a shared data structure named **repository**
- The subsystems are relatively independent, in that they interact only through the repository

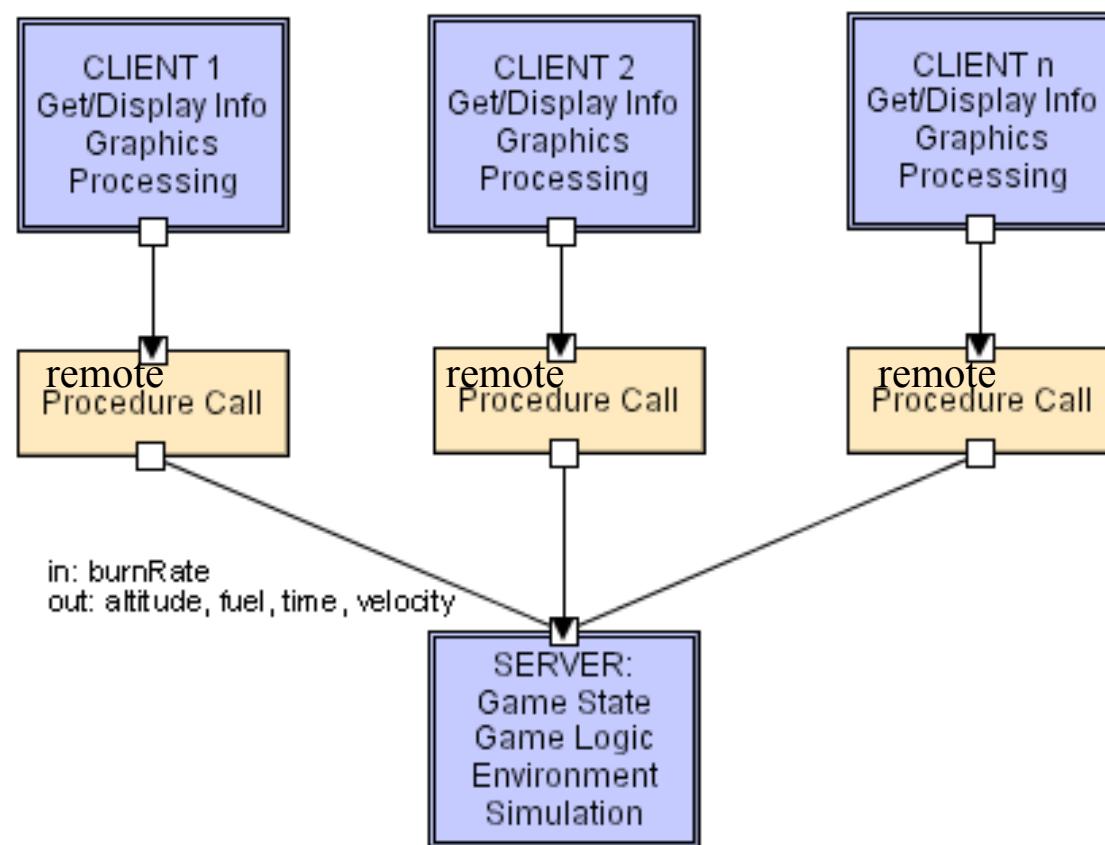


## Example: a compiler architecture based on a repository



# Client-Server style

# Client server

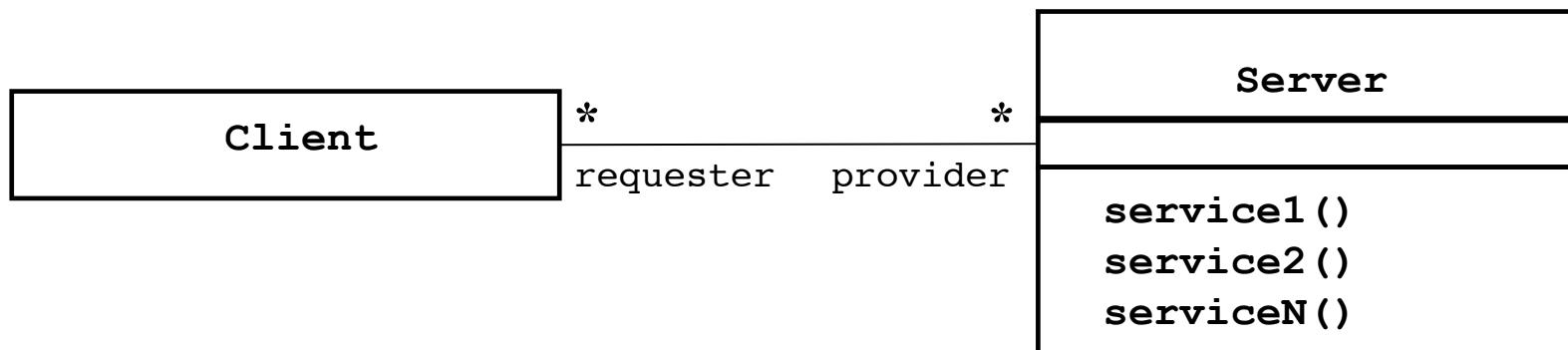


# Client-server paradigm

- It has been conceived in the context of distributed systems, aiming to solve a synchronization issue:
  - A protocol was needed to allow the communication between two different programs
- The driving idea is to make unbalanced the partners' roles within a communication process
  - Reactive entities (named **servers**):
    - They cannot initiate a communication
    - ... they can only answer to incoming requests (reactive entities)
  - Active entities (named **clients**):
    - They trigger the communication process
    - They forward requests to the servers, then they wait for responses

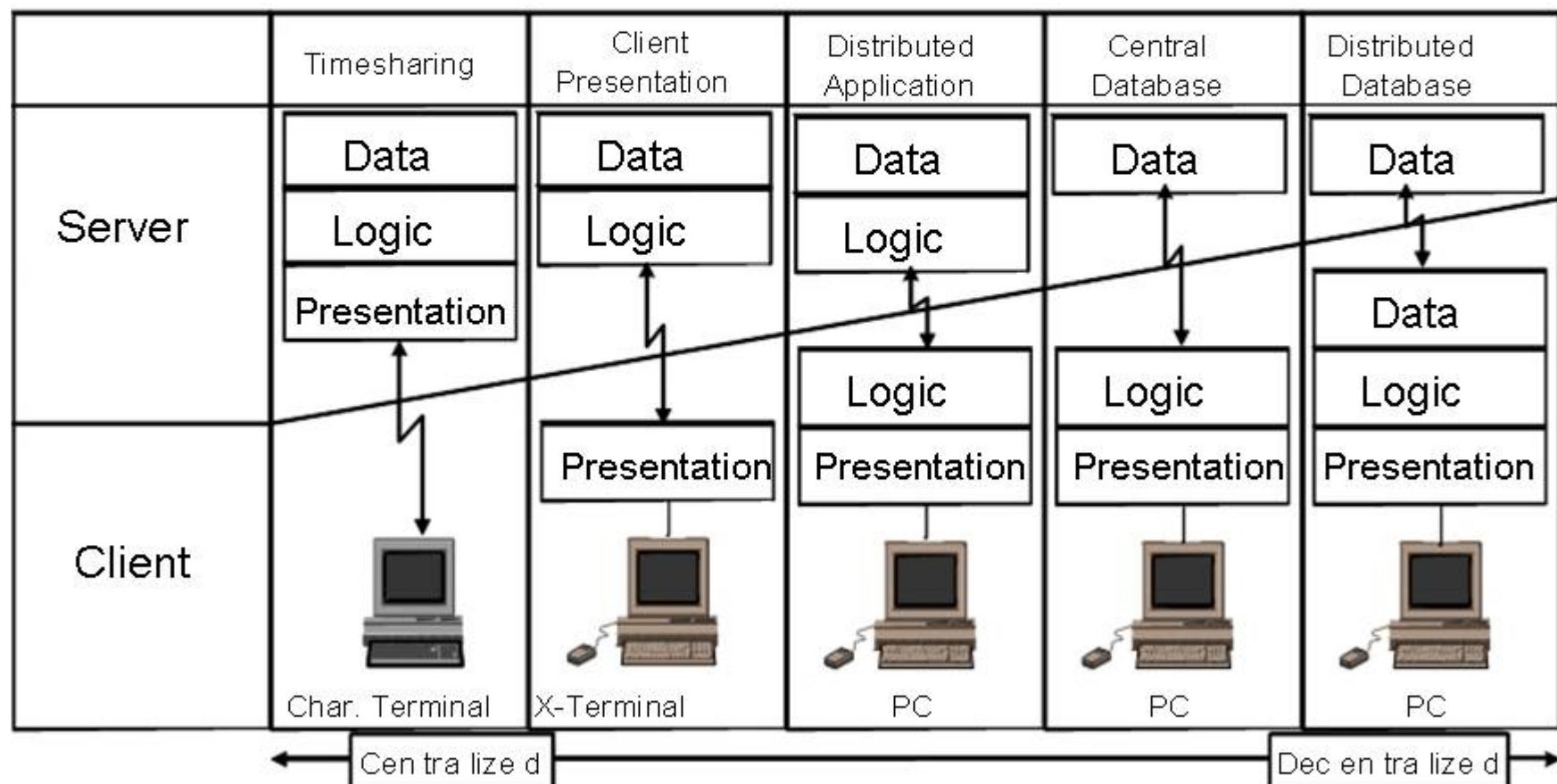
# Client-server style

- From the architectural viewpoint, a client/server system is made up of components which can be classified according to the *service abstraction*:
  - Who provides services is a **server**;
  - Who requires a service is a **client**;
- A client is a program used by a user to communicate with a system
  - ...but a server can be a client for a different service
- Clients are aware of the server interface
- Servers cannot foresee clients' requests



# Client-server style

- 2 tiers Client/Server architectures

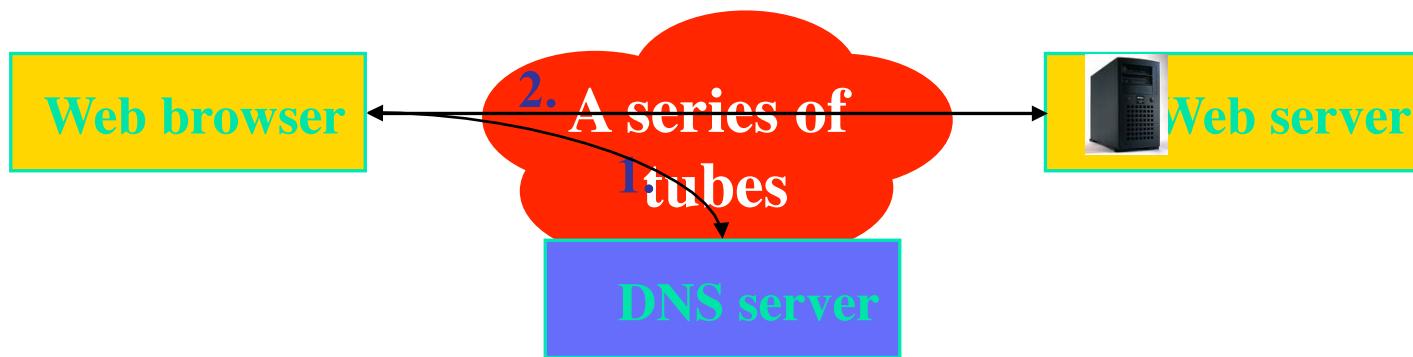


Fat server

Fat client<sup>55</sup>

# Web has a Client-Server style

- HTTP (Hypertext Transfer Protocol), ASCII-based *request/reply protocol* that runs over TCP
  - HTTPS: variant that first establishes symmetrically-encrypted channel via public-key handshake, so suitable for sensitive info



- By convention, servers listen on TCP port 80 (HTTP) or 443 (HTTPS)
- Universal Resource Identifier (URI) format: **scheme**, **host**, **port**, **resource**, **parameters**, **fragment**

`http://search.com:80/img/search/file?t=banana&client=firefox#p2`

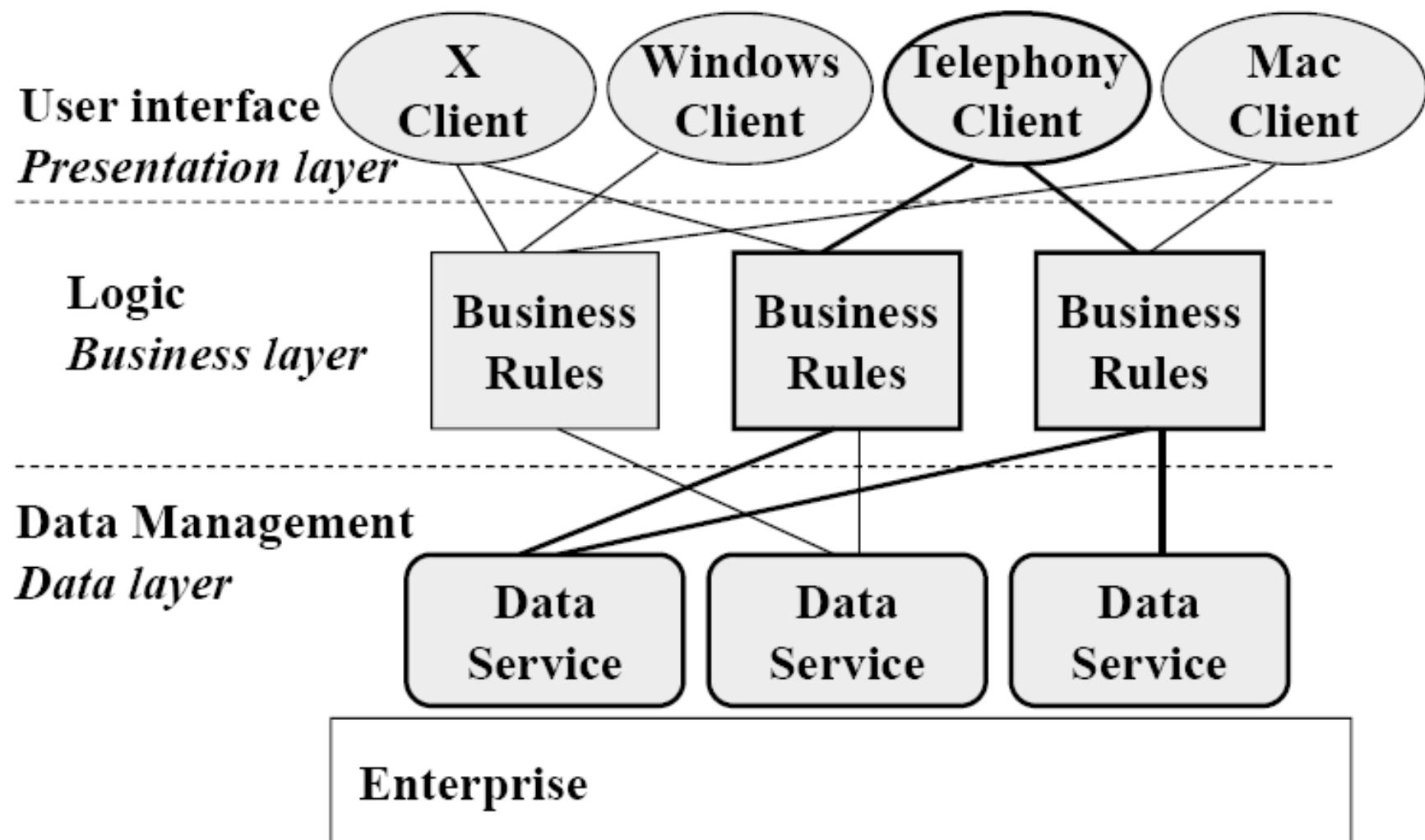
# Three-tier architectures – 1/6

- Early 90's; they propose a clear separation among logics:
  - Tier 1: data management (DBMS, XML files, .....)
  - Tier 2: business logic (application processing, ...)
  - Tier 3: user interface (data presentation and services)
- Each tier has its own goals, as well as specific design constraints
- No assumptions at all about the structure and/or the implementation of the other tiers , i.e.:
  - Tier 2 does not make any assumptions neither on how data are represented, nor on how user interfaces are made
  - Tier 3 does not make any assumptions on how *business logic* works

# Three-tier architectures – 2/6

- Tiers 1 and 3 do not communicate, i.e.:
  - The user interface neither receives any data from data management, nor it can write data
  - Information passing (in both the directions) are filterer by the business logic
- Tiers work as they were not part of a specific application:
  - Applications are conceived as collections of interacting components
  - Each component can take part to several applications at the same time

# Three-tier architectures – 3/6

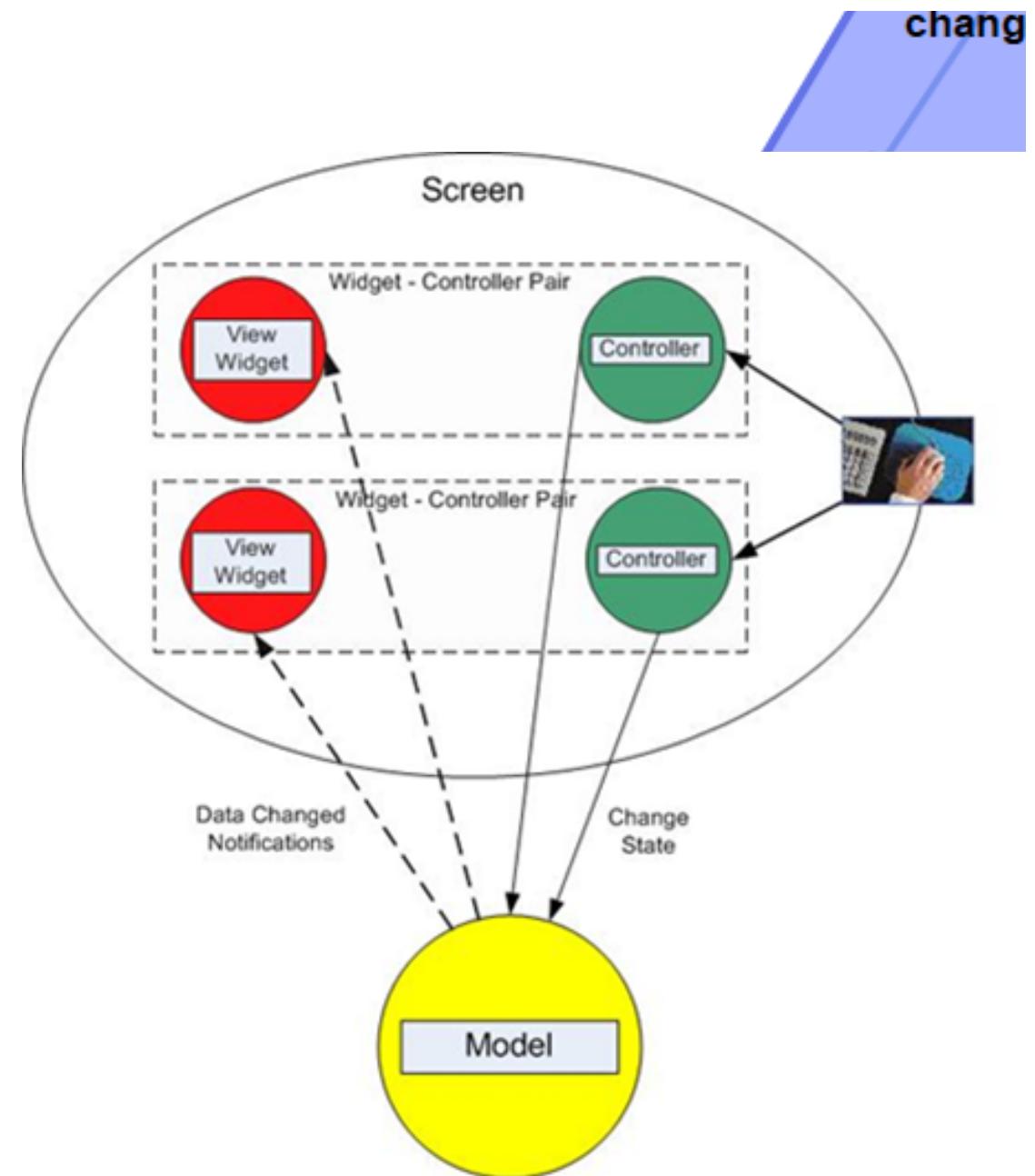


# N-tier architectures

- They provide high flexibility
- Fundamental items:
  - User Interface (UI): e.g., a browser, a WAP minibrowser, a graphical user interface (GUI)
  - Presentation logic, which defines what the UI has to show and how to manage users' requests
  - Business logic, which manages the application business rules
  - Infrastructure services
    - The provides further functionalities to the application components ( messaging, transactions support);
  - Data tier:
    - Application Data level

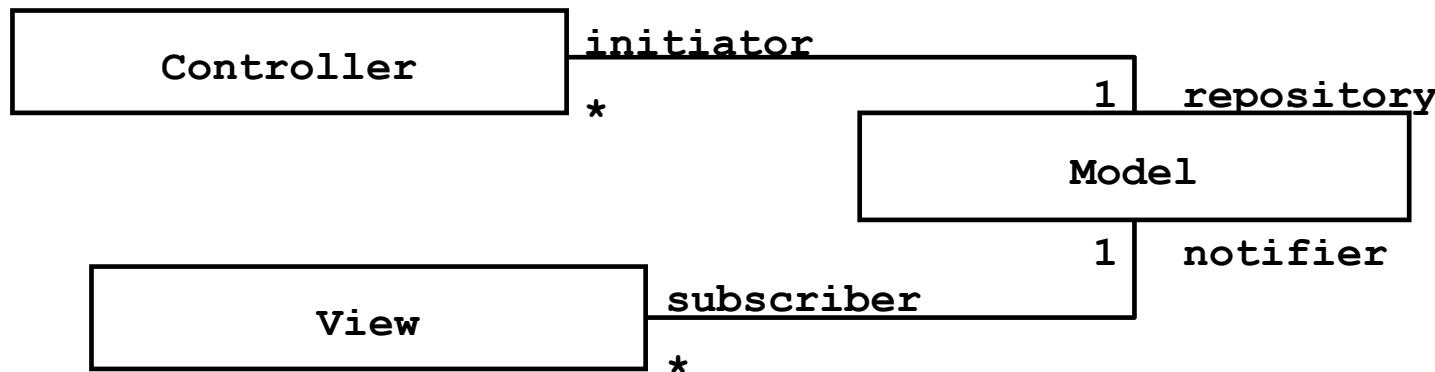
# Model/View/Controller style

# MVC



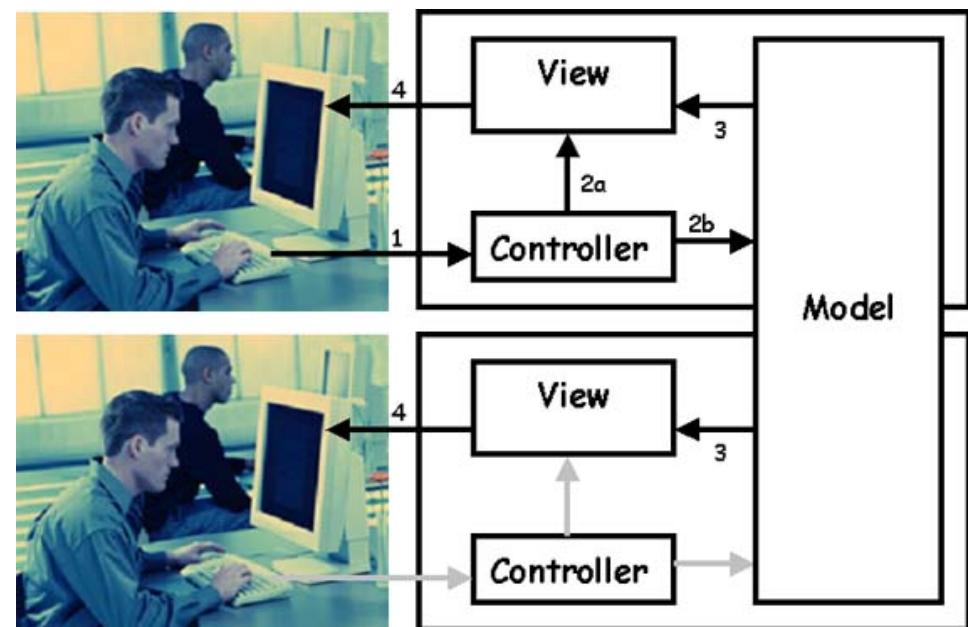
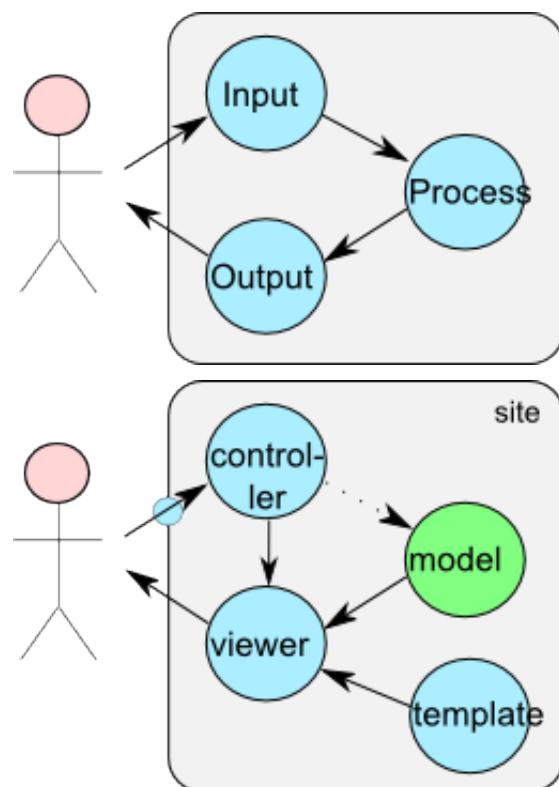
# The MVC style

- Three kinds of subsystems:
  - **Model**, which has the knowledge about the application domain - also called business logic
  - **View**, which takes care of making application objects visible to system users
  - **Controller**, which manages the interactions between the system and its users

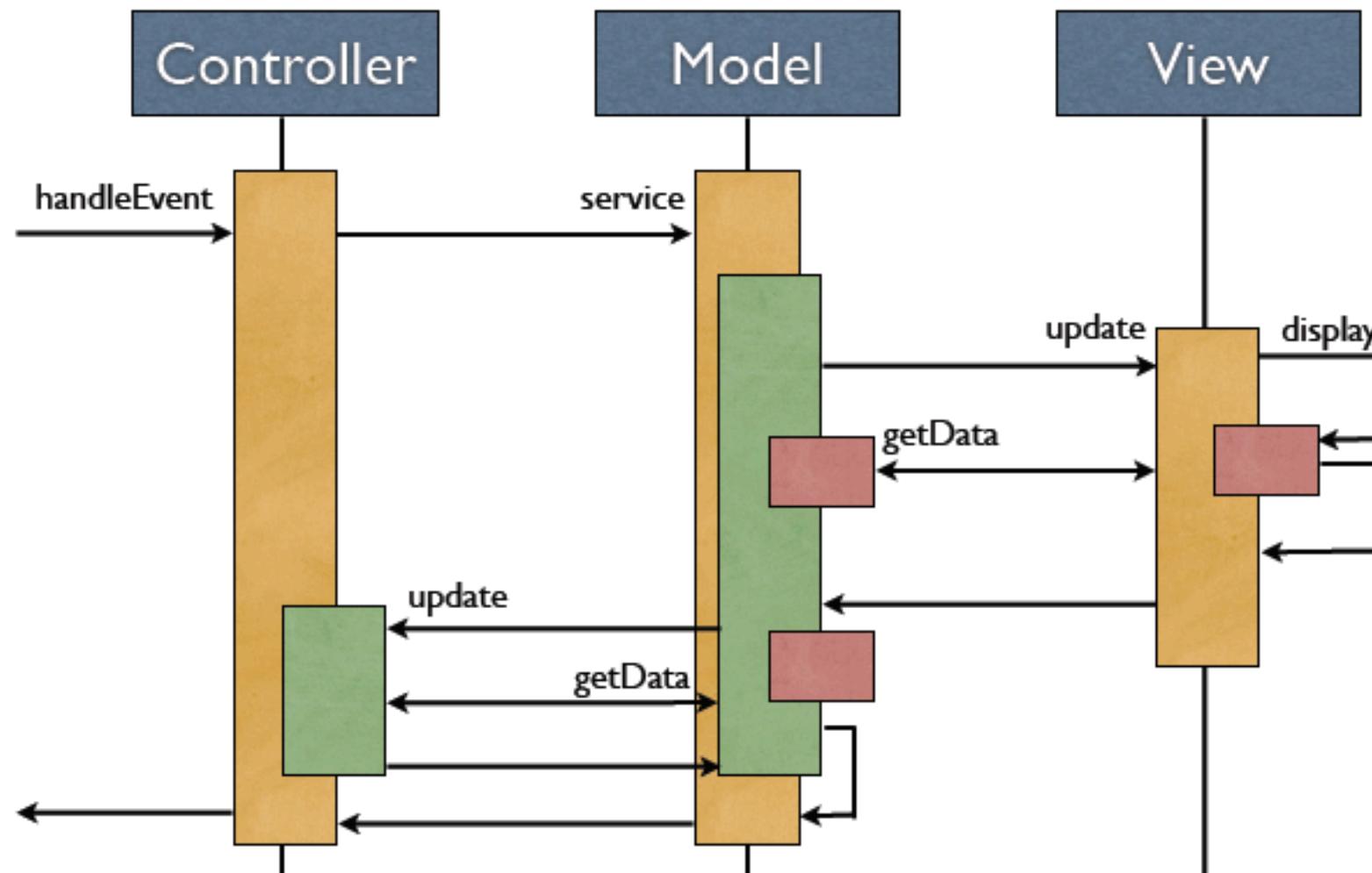


# MVC goal

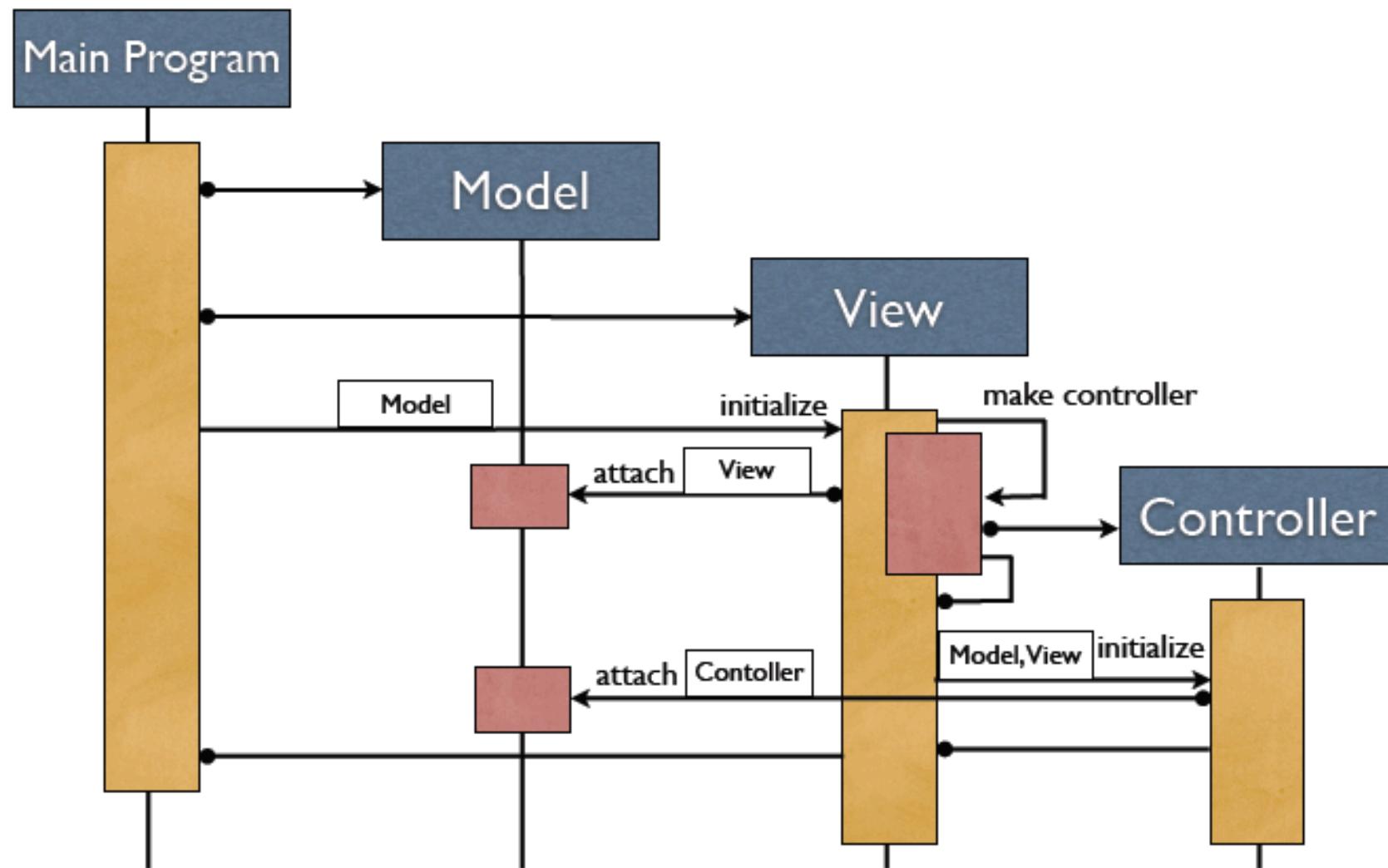
- The goal of MVC is, by decoupling models and views, to reduce the complexity in architectural design and to increase flexibility and maintainability



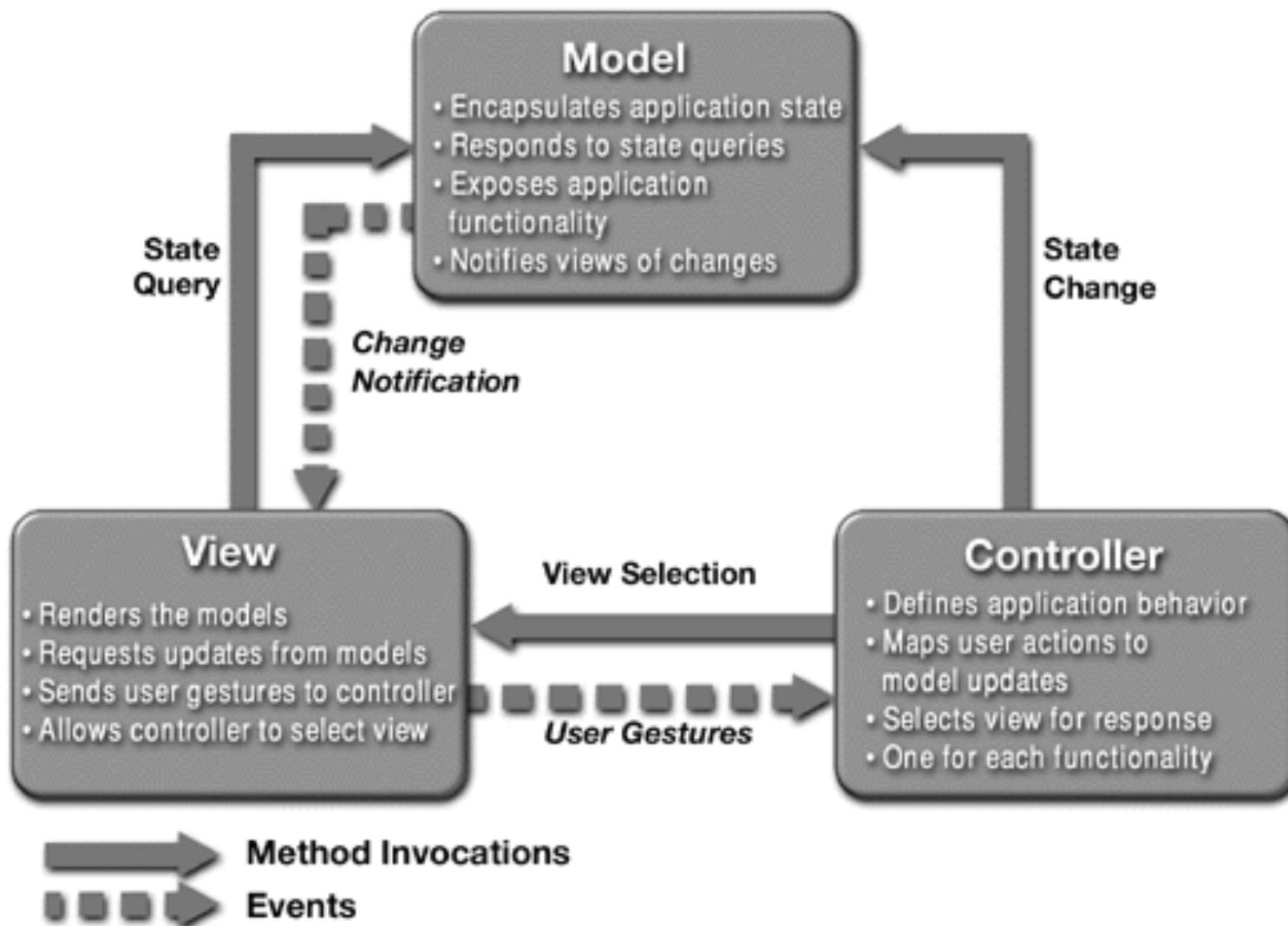
# Basic interaction for MVC



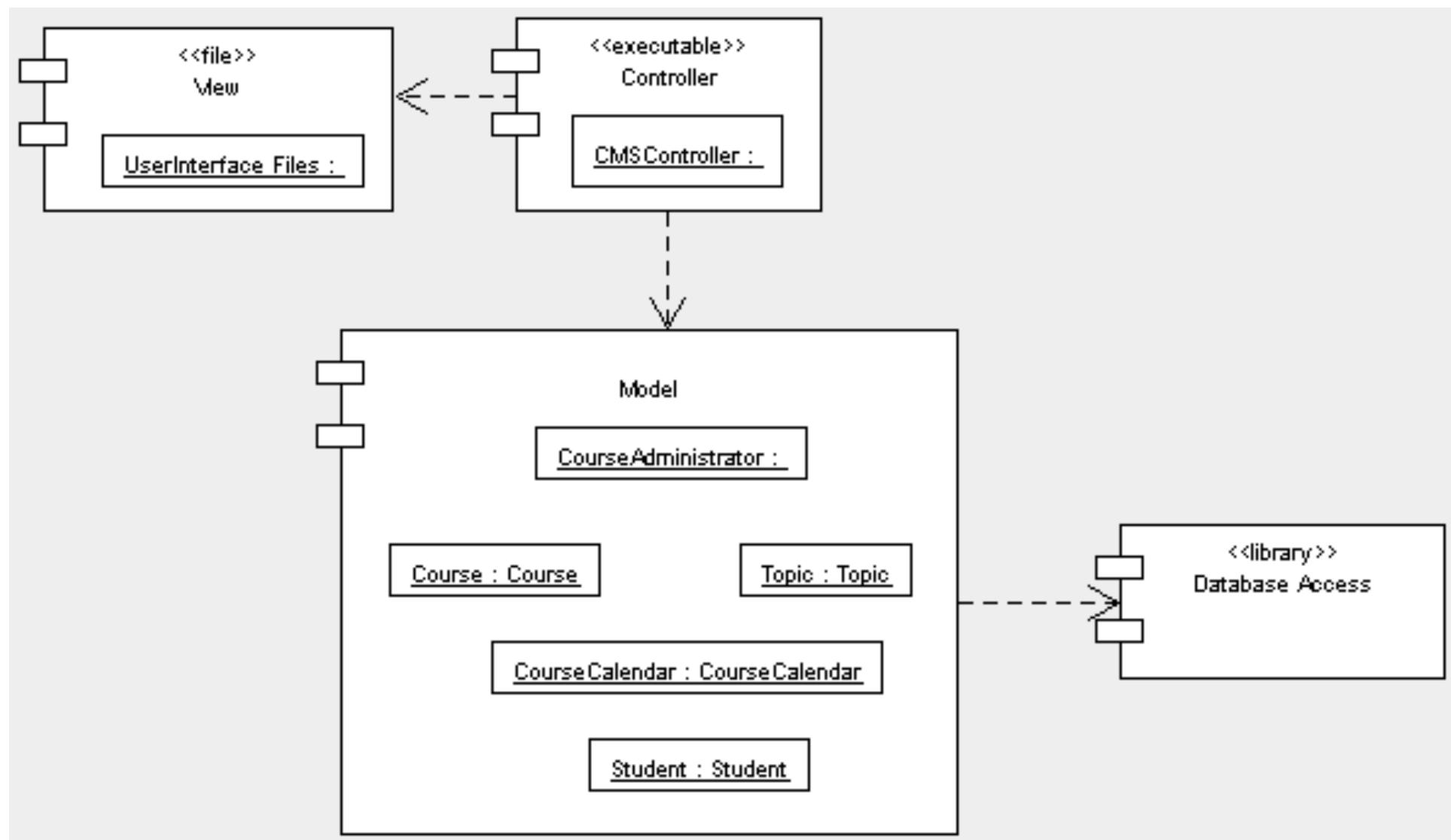
# Initialization for MVC



# MVC interactions



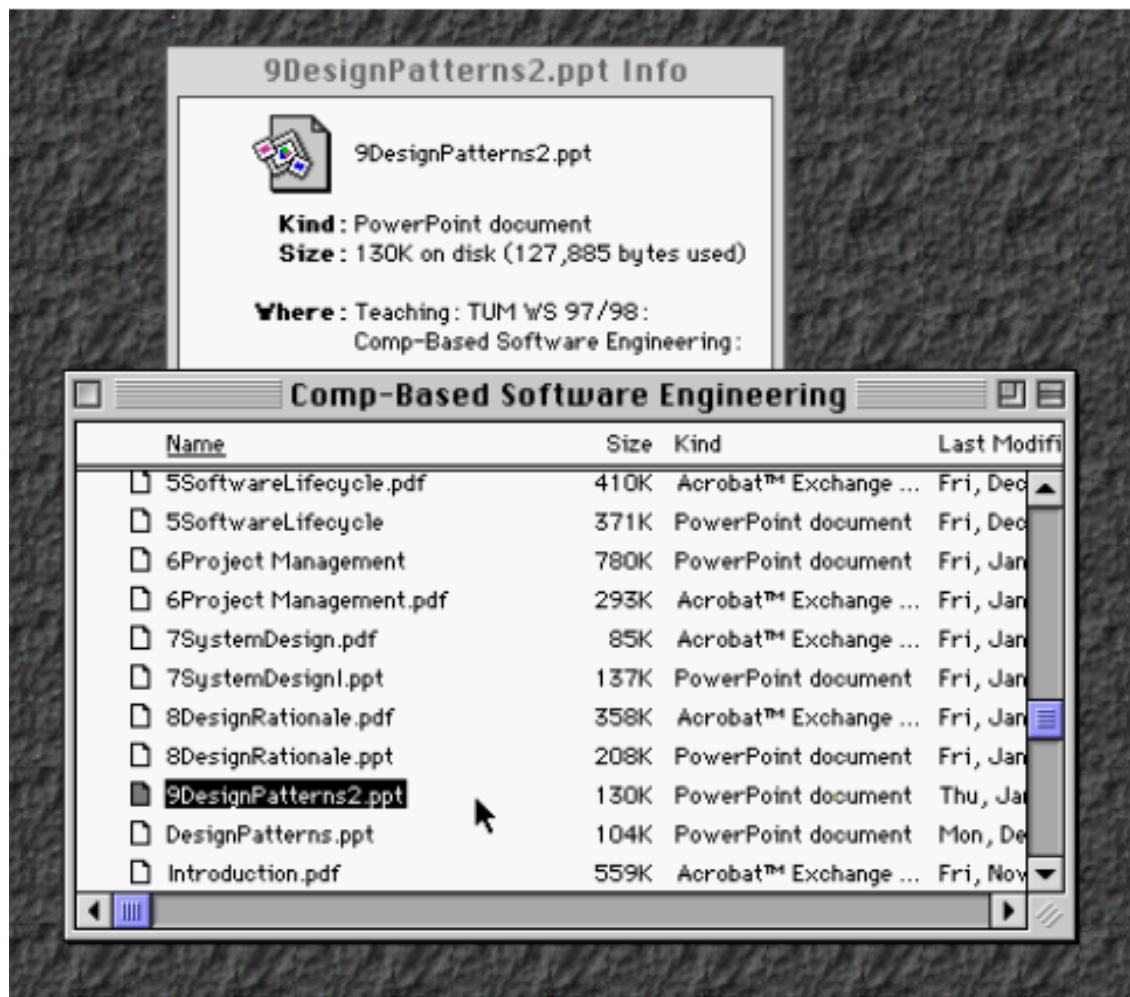
# MVC: component diagram example



# MVC Architectures

- The Model does not depend on any View and/or Controller
- Changes to the Model state are communicated to the View subsystems by means of a “subscribe/notify” protocol (eg. Observer pattern)
- MVC is a combination of the repository and three-tiers architectures:
  - The Model implements a centralized data structure;
  - The Controller manages the control flow: it receives inputs from users and forwards messages to the Model
  - The Viewer pictures the model

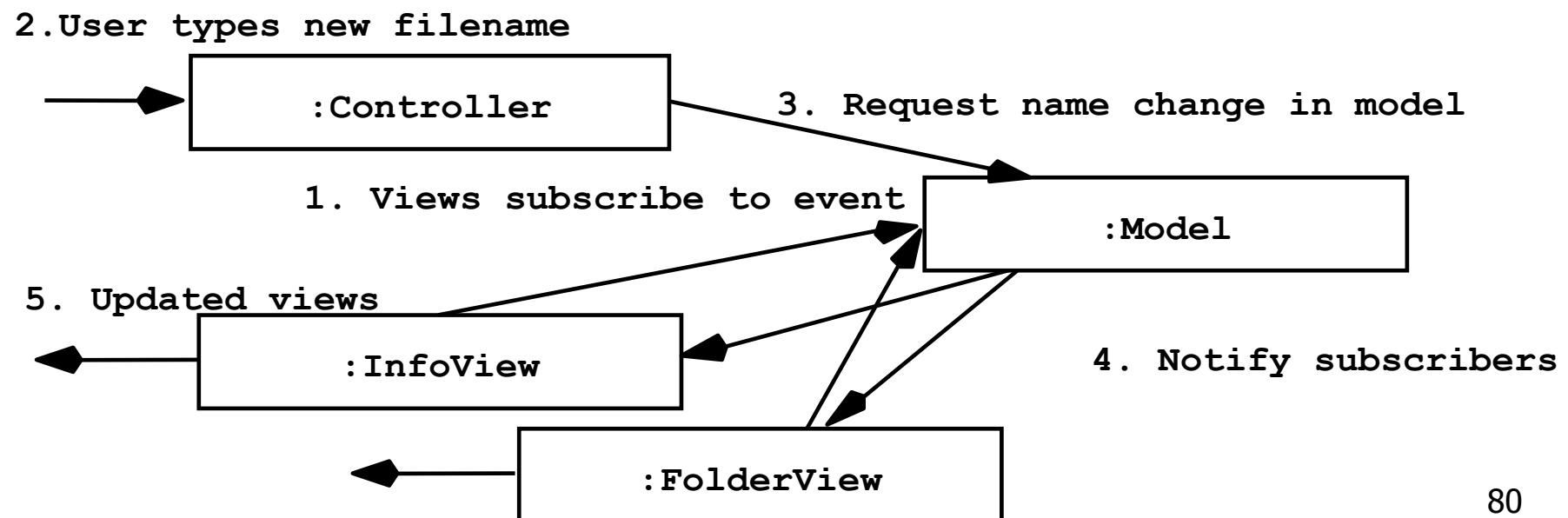
# MVC: An example



- Two different views of a file system:
- The bottom window visualizes a folder named Comp-Based Software Engineering
- The up window visualizes file info related to file named 90DesignPatterns2.ppt
- The file name is visualized into three different places

# MVC: communication diagram

- 1. Both InfoView and FolderView subscribe the changes to the model File when created
- 2. The user enters the new filename
- 3. The Controller forwards the request to the model
- 4. The model actually changes the filename and notifies the operation to subscribers
- 5. InfoView and FolderView are updated so that the user perceives the changes in a consistent way



# ClientServer 3-tiers vs MVC

- The CS 3-tiers style may seem similar to the model-view-controller (MVC) style; however, they are different
- A fundamental rule in a 3-tiers architecture is: the client tier **never** communicates directly with the data tier; all communication must go through the middleware tier: the 3-tiers architecture is linear
- Instead, the MVC architecture is triangular: the View sends updates to the Controller, the Controller updates the Model, and the Views get updated directly from the Model

# The benefits of the MVC style

- The main reason behind the separation (Model, View and Controller) is that the user interfaces (Views) are changed much more frequently than the knowledge about the application domain (Model)
- This style is easy to reuse, to maintain, and to extend
- The MVC style is especially effective in the case of interactive systems, when multiple synchronous views of the same model have to be provided

# Service-Oriented style

## (SOA)

# Service-oriented architectures

- Service-oriented architecture (SOA) is an approach to loosely coupled, protocol independent, standards-based distributed computing where a **software resource available on the network** is considered as a **service**
- SOA is a technology solution that provides an enterprise with the agility and flexibility its users have been looking for
- The integration process is based on a composition of services spanning multiple enterprises

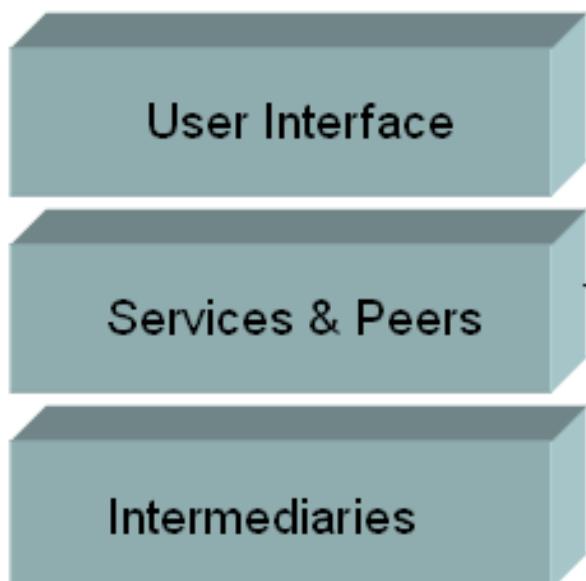
# Service-oriented architectures

- The following principles define the rules for development, maintenance, and usage of a SOA:
  - Reuse, granularity, modularity, composability, componentization, and interoperability
  - Compliance to standards (either cross-industry or industry-specific)
  - Services identification and categorization, provisioning and delivery, and monitoring and tracking

# Service computing: layered style

**The three layered views of service computing...**

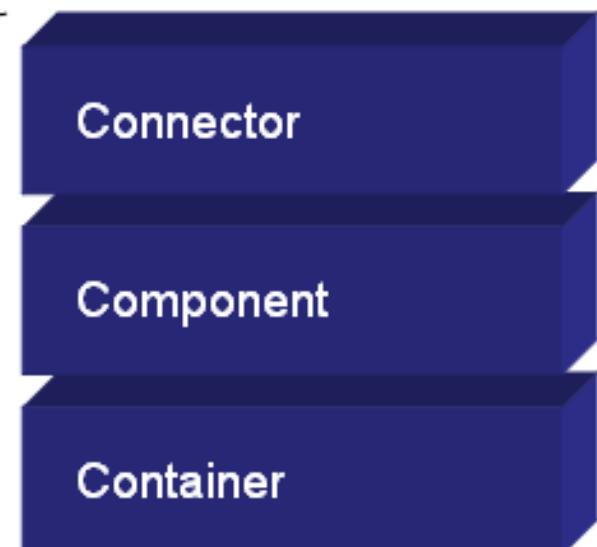
A Layered Service Network View



A Layered Services View



A Layered Service View



Copyright 2005 MomentumSI.

# SOA principles – 1/2

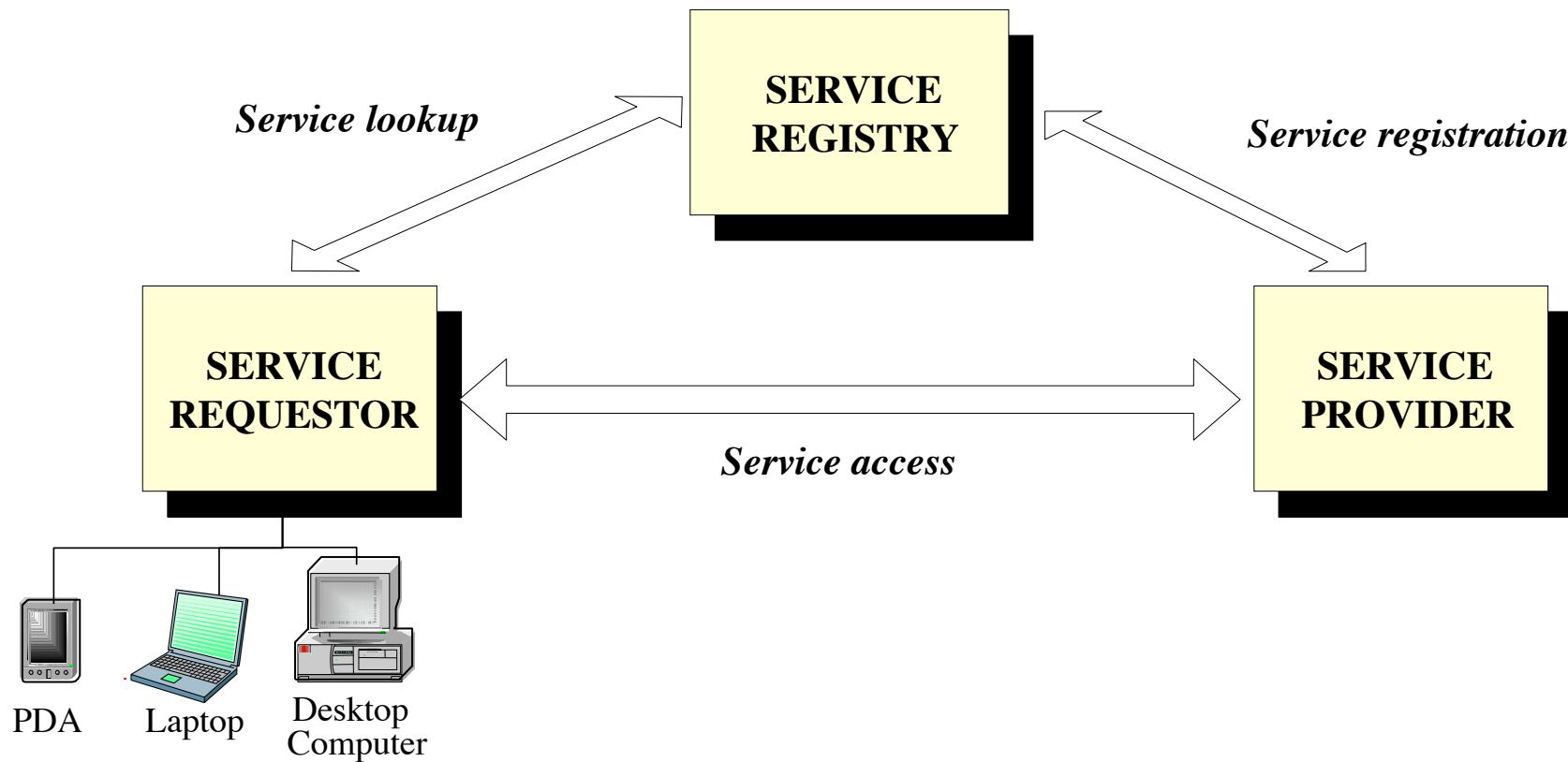
- Service **encapsulation** - Many web-services are consolidated to be used under the SOA Architecture. Often such services have not been planned to be under SOA
- Service **loose coupling** - Services maintain a relationship that minimizes dependencies and only requires that they maintain an awareness of each other
- Service **contract** - Services adhere to a communications agreement, as defined collectively by one or more service description documents
- Service **abstraction** - Beyond what is described in the service contract, services hide logic from the outside world

# SOA building blocks – 1/2

- Service **Consumer** (also known as Service **Requestor**): it locates entries in the Registry using various find operations and then binds to the service provider in order to invoke one of its services
- Service **Provider**: it creates a service and publishes its interface and access information to the Service registry
- Service **Registry** (also known as Service **Broker**): it is responsible for making the Service interface and implementation access information available to any potential service requestor

# SOA building blocks – 2/2

- Service registration (provider) and lookup (requestor) in the registry
- Service access



# SOA characteristics

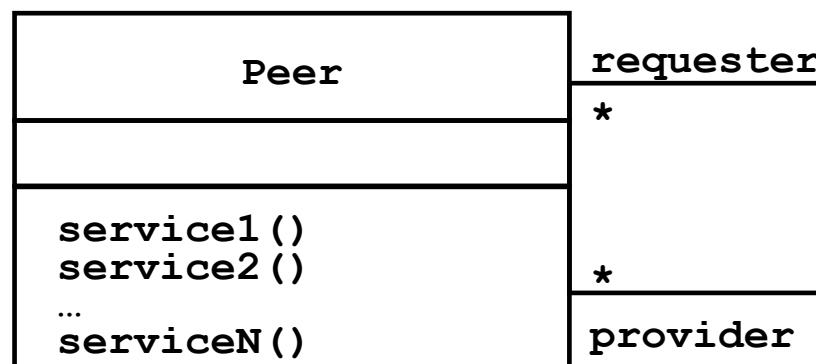
- The software components in a SOA are services based on standard protocols
- Services in SOA have minimum amount of interdependencies
- Communication infrastructure used within an SOA should be designed to be independent of the underlying protocol layer
- Offers coarse-grained business services, as opposed to fine-grained software-oriented function calls
- Uses service granularity to provide effective composition, encapsulation and management of services

# Peer-to-peer style

P2P

# Peer-to-peer architectures

- They can be considered a generalization of the client/server architecture
- Each subsystem is able to provide services, as well as to make requests
  - Each subsystem acts both as a server and as a client



# Peer-to-peer architectures

- In such an architecture, all nodes have the same abilities, as well as the same responsibilities. All communications are (potentially) bidirectional
- The goal:
  - To share resources and services (data, CPU cycles, disk space, ...)
- P2P systems are characterized by:
  - Decentralized control
  - Adaptability
  - Self-organization and self-management capabilities

# Peer-to-peer architectures

- Typical functional characteristics of P2P systems
  - File sharing system
  - File storage system
  - Distributed file system
  - Redundant storage
  - Distributed computation
- Typical non-functional requirements
  - Availability
  - Reliability
  - Performance
  - Scalability
  - Anonymity

# The phases of a P2P application

A P2P application is organized in three phases:

- **Boot**, during which a peer connects to the network and actually performs the connections (P2P boot is rare)
- **Lookup**, during which a peer looks for a provider of a given service or information (generally providers are SuperPeers)
- **Resource sharing**, during which requested resources are delivered, usually in several segments

# P2P classification – 1/2

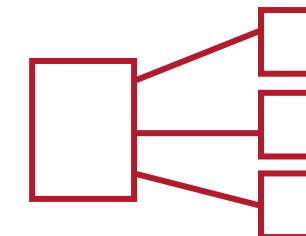
- P2P applications can be categorized as follows:
  - **Pure P2P applications**
    - All the phases are based on P2P
  - **P2P applications**
    - lookup and resource sharing are performed according to the P2P paradigm;
    - Some servers are used to make the boot
  - **Hybrid P2P applications :**
    - The resource sharing is performed according to the P2P paradigm
    - Some servers are used to make the boot;
    - Specific peers are used to perform lookup.

# P2P classification – 2/2

- With respect to lookup phase:

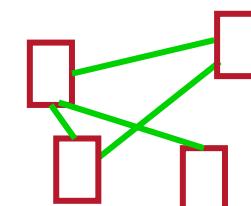
- Centralized Lookup

- Centralized index of providers



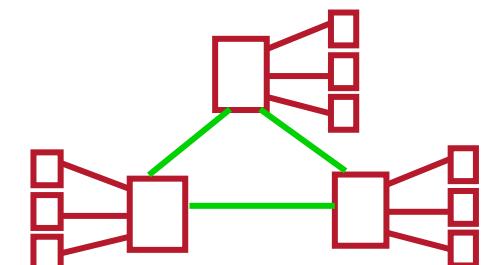
- Decentralized Lookup

- Distributed index



- Hybrid Lookup

- Several centralized systems which are linked into a decentralized system



# Software Testing

# Outline

- Introduction to Testing
  - Definition
  - Rationale
- Some Types of Testing
  - Black-Box, White-Box, Reliability, Security, Regression Testing
- Test Process Management
  - V Model - for Classical Software Development:
  - TDD – for Agile Software Development:
- Unit Testing
- Acceptance Testing
- Formal Verification

# Definition of Software Testing

- Software Testing is the process of
  - Evaluating an attribute or capability of a program or system
  - Determining that a system meets its requirements
  - Finding **defects**
  - Exploring and understanding the status of the benefits and risks associated with the **release** of a software system

# Some Different Types of Testing

- Acceptance Testing
- Black Box Testing
- White Box Testing
- Compatibility Testing
- Conformance Testing
- Functional Testing
- Integration Testing
- Load Testing
- Performance Testing
- Regression Testing
- Smoke Testing
- Stress Testing
- Security Testing
- Unit Testing

*This isn't all of it, there is a lot more*

# Why Test?



- Q: If all software is released to customers with faults, why should we spend so much time, effort, and money on testing?
  - Considering the **potential costs of Delaying the Release** of a Software Product
  - **New products:** The first to the market often sells better than superior products that are released later.

# Cost to fix faults

- It is commonly believed that the earlier a defect is found the cheaper it is to fix it.

Cost to fix a defect		Time detected				
		Requirements	Architecture	Construction	System test	Post-release
Time introduced	Requirements	1×	3×	5 – 10×	10×	10 – 100×
	Architecture	-	1×	10×	15×	25 – 100×
	Construction	-	-	1×	10×	10 – 25×

# Beta Testing

- Beta Testing
  - Customers test for free!
  - Seems to give you test cases representative of customer use.
  - Helps to determine what is most important to the customers.
  - Can do more configuration (environment) testing than in your testing lab.
- Disadvantages:
  - Beta testers might have a particular perspective to the system, which result in not catching diverse system bugs.
  - Beta testers usually won't report: usability problems, bugs they don't understand and bugs that seem obvious.
  - Most beta testers are sometimes "techies" who have a higher tolerance of bugs. They do not represent the average customer.
  - Takes much more time and effort to handle a user reported bug.

# General Testing Approaches

*High-level approaches to testing that can be applied to the specific testing techniques*

- **Positive and Negative Testing**
  - **Positive testing**
    - Is intended to verify that a software system conforms to its stated requirements
    - Does the system “*fit for purpose*”?
  - **Negative Testing**
    - Is intended to assure that the system does NOT do what it is supposed to do
- **Black Box and White Box Testing**
- **Experienced Based Testing**
  - Is intended to design and perform effective tests
  - As suggested by its name, the skill is based on the experience

# General Testing Approaches:

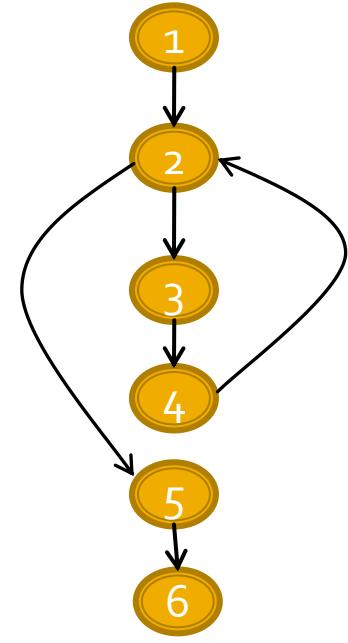
## Black Box Testing

- Used to test **functional parts** of a program.
- The tester **has no prior knowledge** to the internal workings of the software other than what it is supposed to output.
- Advantages
  - **Unbiased**, no programming knowledge needed
  - Test cases can be made very early on after specs are done.
- Disadvantages
  - Can't identify all possible test cases
  - Can be redundant and cannot possibly test every single possible path.



# General Testing Approaches: White Box Testing

- Used to test the **structural parts** of the software.
- The **tester has explicit knowledge** about the **internal workings** of the software and the programming language its written in.
- Knowing how the software is supposed to work, the tester chooses **specific paths** and determines the appropriate output.
- While white-box testing can be applied at the **unit**, **integration** and **system** levels of the software testing process, it is usually done at the unit level



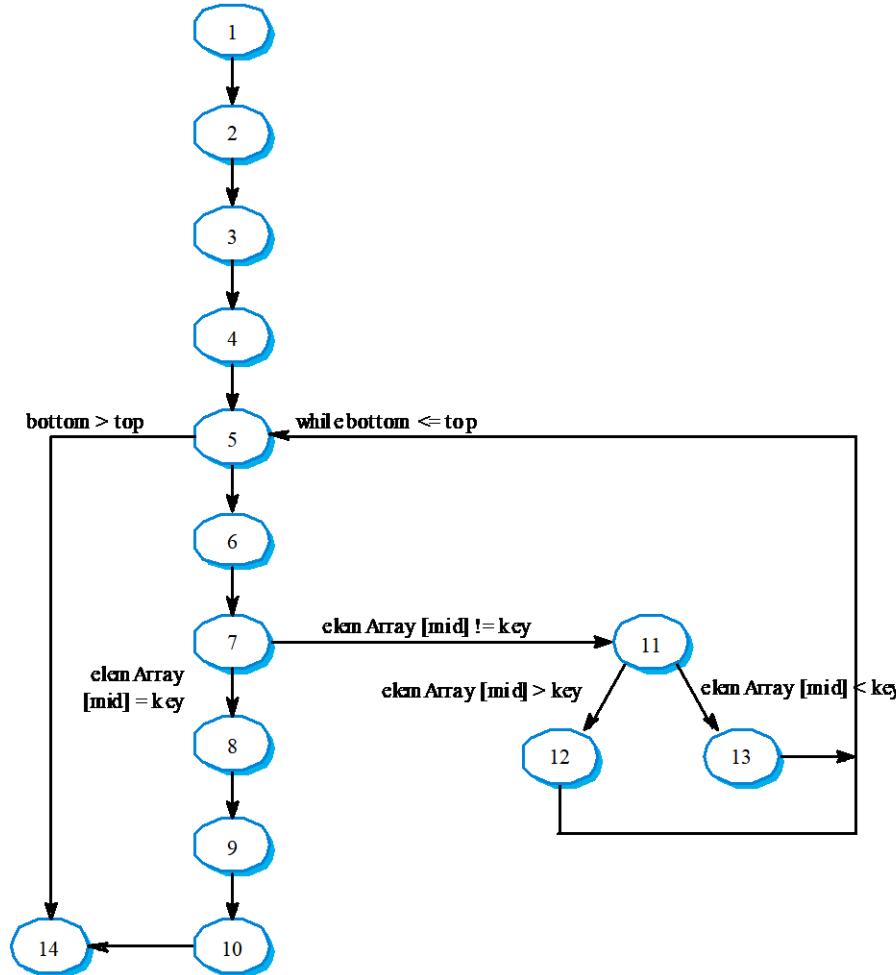
program flow graph

# White Box Testing Example:

## Binary search pseudo-code

```
class BinSearch {  
  
    // This is an encapsulation of a binary search function that takes an array of  
    // ordered objects and a key and returns an object with 2 attributes namely  
    // index - the value of the array index  
    // found - a boolean indicating whether or not the key is in the array  
    // An object is returned because it is not possible in Java to pass basic types  
    // by  
    // reference to a function and so return two values  
    // the key is -1 if the element is not found  
  
        public static void search ( int key, int [] elemArray, Result r )  
    {  
        1.      int bottom = 0 ;  
        2.      int top = elemArray.length - 1 ;  
        3.      int mid ;  
        4.      r.found = false ;  
        5.      r.index = -1 ;  
        6.      while ( bottom <= top )  
        {  
            7.          mid = (top + bottom) / 2 ;  
            8.          if (elemArray [mid] == key)  
            {  
                9.              r.index = mid ;  
                10.             r.found = true ;  
                11.             return ;  
            } // if part  
            12.            else  
            {  
                13.                if (elemArray [mid] < key)  
                    bottom = mid + 1 ;  
                else  
                    top = mid - 1 ;  
            }  
        } //while loop  
    14.    } // search  
} //BinSearch
```

# Binary search flow graph



# White Box Testing

*example – Cont.*

- Independent Paths
  - 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 14
  - 1, 2, 3, 4, 5, 14
  - 1, 2, 3, 4, 5, 6, 7, 11, 12, 5, ...
  - 1, 2, 3, 4, 6, 7, 2, 11, 13, 5, ...
- **Test cases** should be derived so that all of these paths are executed
- **Advantages:**
  - Helps with optimizing code
  - Can be very thorough
- **Disadvantages:**
  - Cannot detect missing parts of the specifications or missing requirements.
  - State explosion problem

# Specific Testing Techniques

- Performance Testing
  - Load Testing
  - Stress Testing
  - Endurance Testing
- Security Testing
  - Confidentiality
  - Integrity
  - Authentication
  - Availability
  - Authorization
  - Non-repudiation
- Regression Testing
- Unit Testing
- Integration Testing
- System Testing

# Performance Testing

*Software performance testing is used to determine the speed or effectiveness of a computer, network, software program or device under particular circumstances*

- Performance testing is used to make sure that because of the software, system performance should not degrade.
- Sometimes, specifications don't mention how optimal they want the software to be.
  - Should not take an infinite amount of time or infinite resources to run.
- Benchmarks!
  - Used to test performance on a typical system.
  - Gives averages on how a low-end and high-end system would run given the software.

# Performance Testing

- **Load Testing**
  - Understand the behavior of the application under a **specific expected load**
  - A load can be expected concurrent number of users on the application
  - Highlights any **bottlenecks** in the application software (e.g. DB, App Server)
- **Stress Testing**
  - Understand the **upper limits of capacity** within the application landscape.
  - Determine the application's **robustness** in terms of extreme load
  - Determine if the application will perform sufficiently if the current load goes well above the expected maximum
- **Endurance Testing**
  - Determine if the application can **sustain** the continuous expected load
  - Find out **performance degradation**
    - To ensure that the throughput and/or response times after some long period of sustained activity are as good or better than at the beginning of the test

# Security Testing

- Software flaws leave huge gaping holes for hackers.
  - Companies worldwide often face security attacks.
  - Having high quality and reliable code helps a lot.
- Needs to cover:
  - Confidentiality
  - Integrity
  - Authentication
  - Availability
  - Authorization
  - Non-repudiation

# Regression Testing

**Software Regression:** is a software bug which makes a feature stop functioning as intended after a certain event (e.g. a system upgrade)

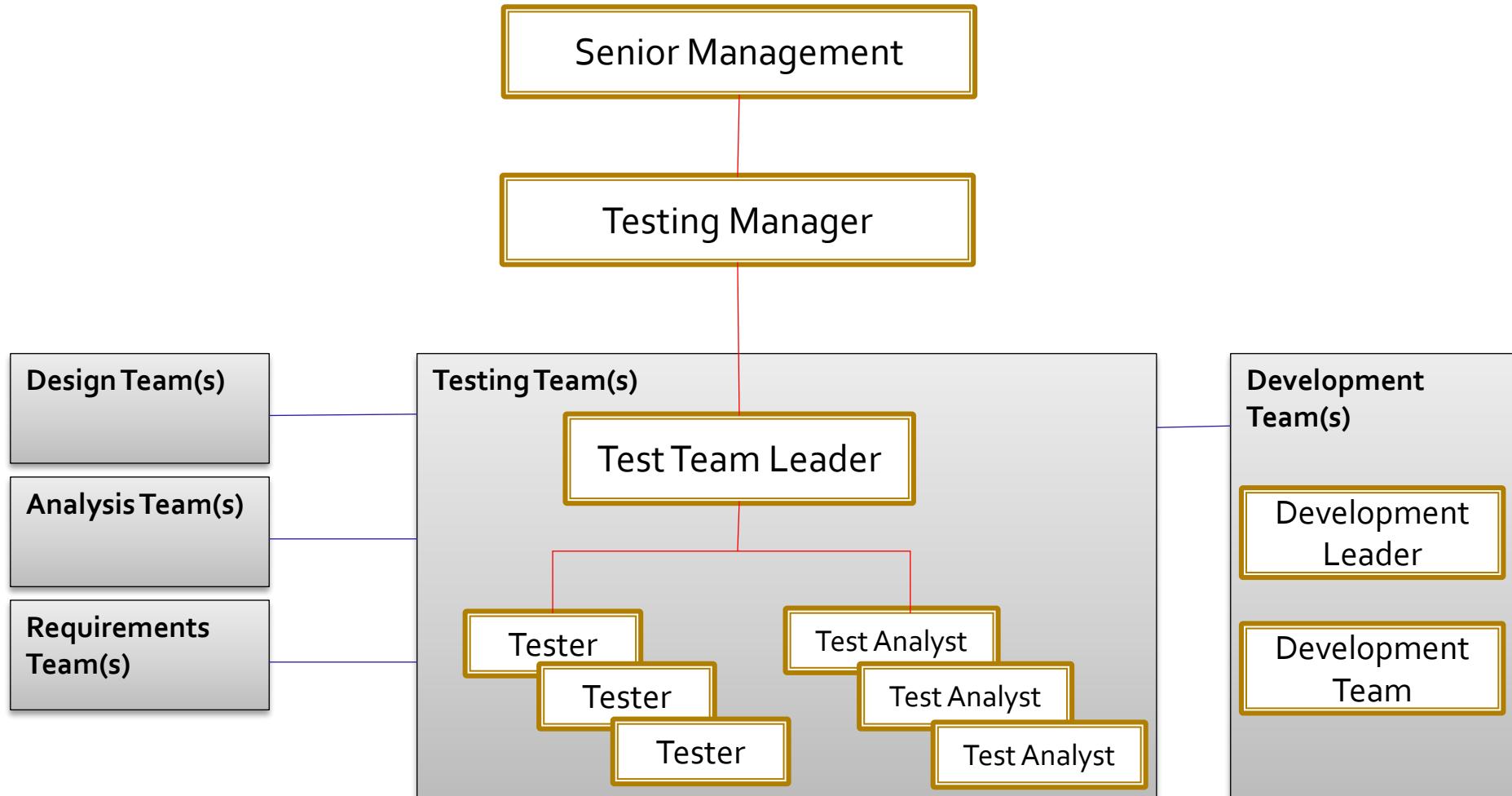
- When you make new changes to software.
  - The change may break in your program.
  - An old bug which you may have fixed arises again.
- **Regression testing** is a quality control method where it:
  - Checks to see whether new code will compile.
  - Checks to see if the unmodified code still works.
  - Can be (and is strongly recommended to be) automated
- Regression testing is an integral part of the **XP**
  - Design documents are partly replaced by extensive, repeatable, and automated testing of the entire software package at every stage of development
- **Advantages**
  - Since you test every time you add something new, you can easily find what caused the error.
- **Disadvantages**
  - In large scale projects, doing a regression test could take a long time.

# Test Process management

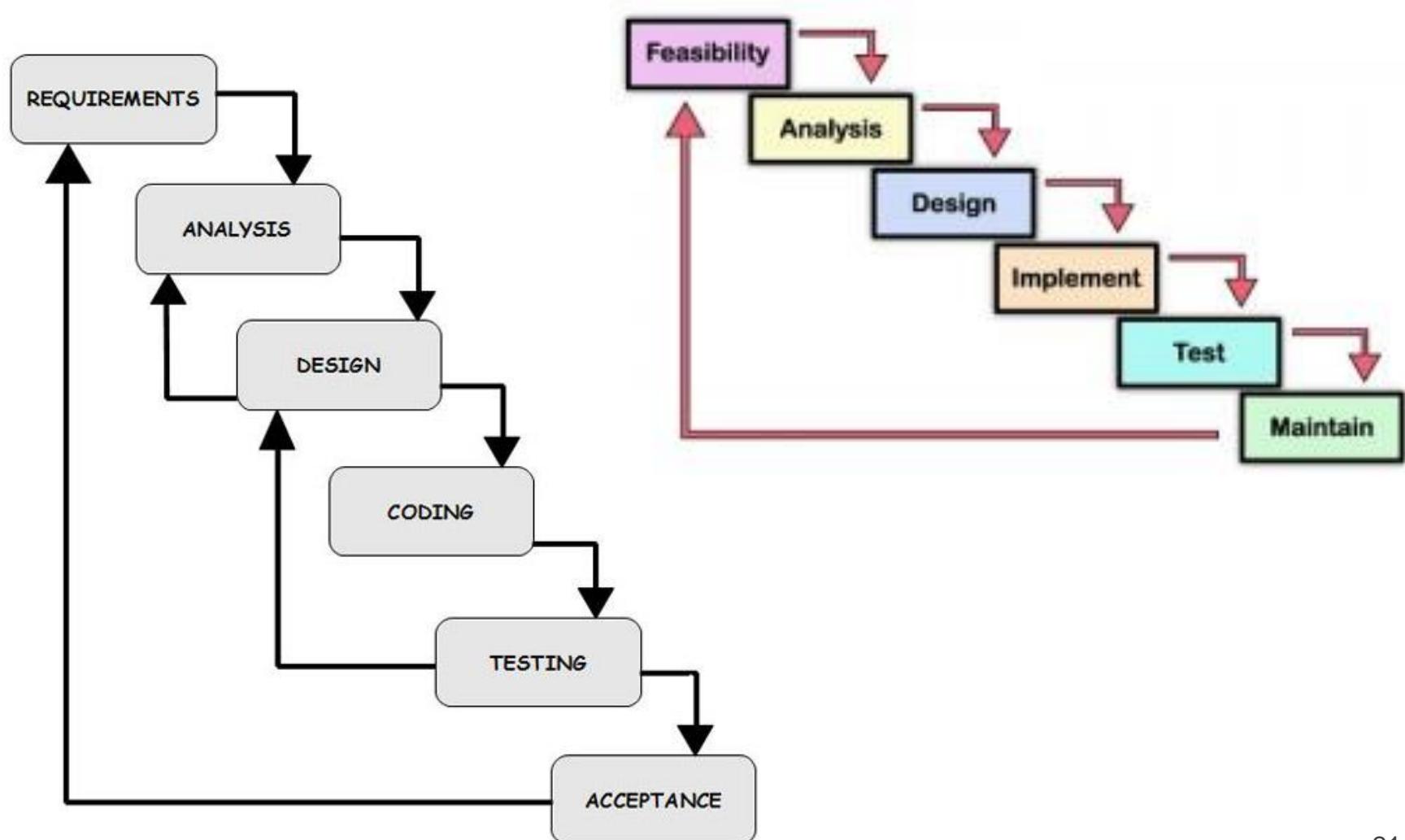
- Classical Software Development
  - Testing in Waterfall development
  - Testing in V-Model
  - Testing in RUP
- Agile Software Development
  - TDD – Test Driven Development

# Management and Planning of Testing

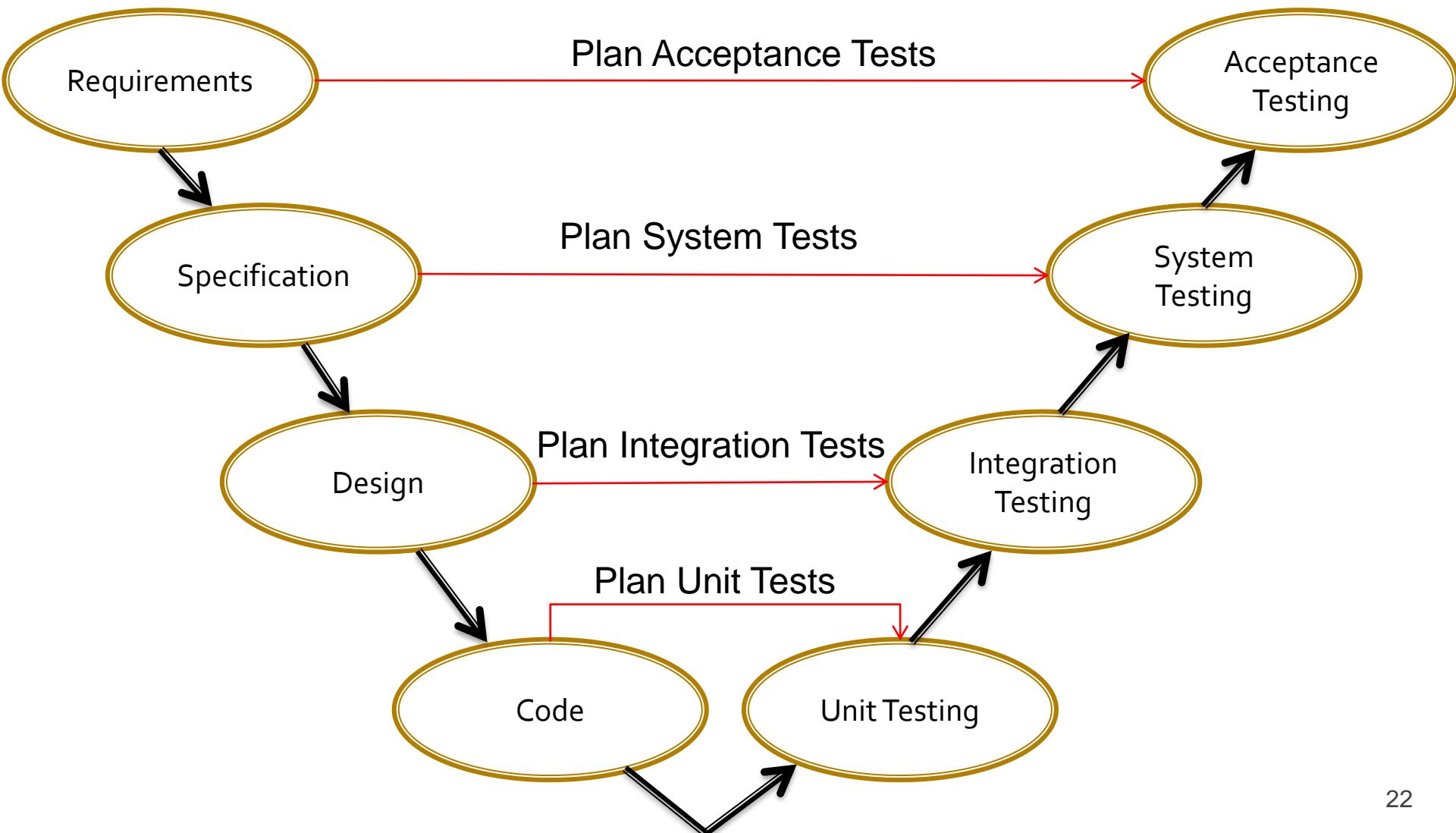
*classical approach*



# Testing in Waterfall model



# Testing in V-Model

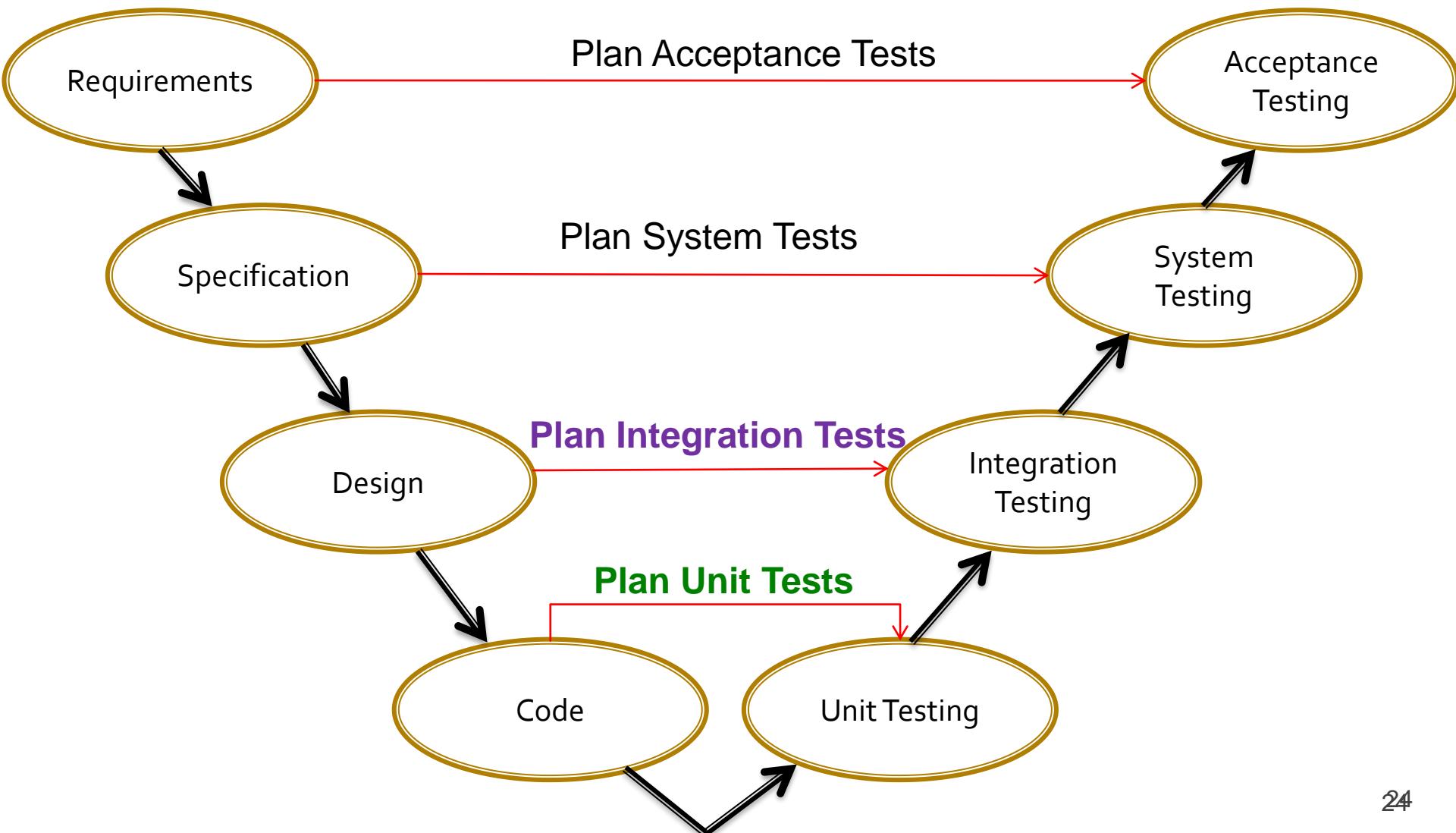


# Unit Testing

*Testing individual units of source code to determine if they are fit for use.*

- What is a **unit of code**?
  - A unit is the smallest testable part of an application
- Unit tests are typically **written and run by software developers** to ensure that code meets its design and behaves as intended
- A unit test provides a strict, written **contract** that the piece of code must satisfy
- Frameworks:
  - Started with SUnit for Smalltalk, written by Kent Beck
  - You're probably familiar with JUnit.
- **Benefits of Unit Testing:**
  - To isolate each part of the program and show that the individual parts are correct
  - Facilitates Change
  - Simplifies Integration
  - Documentation

# Testing in V-Model



# Integration Testing

*Combining units of system and testing them as a group*

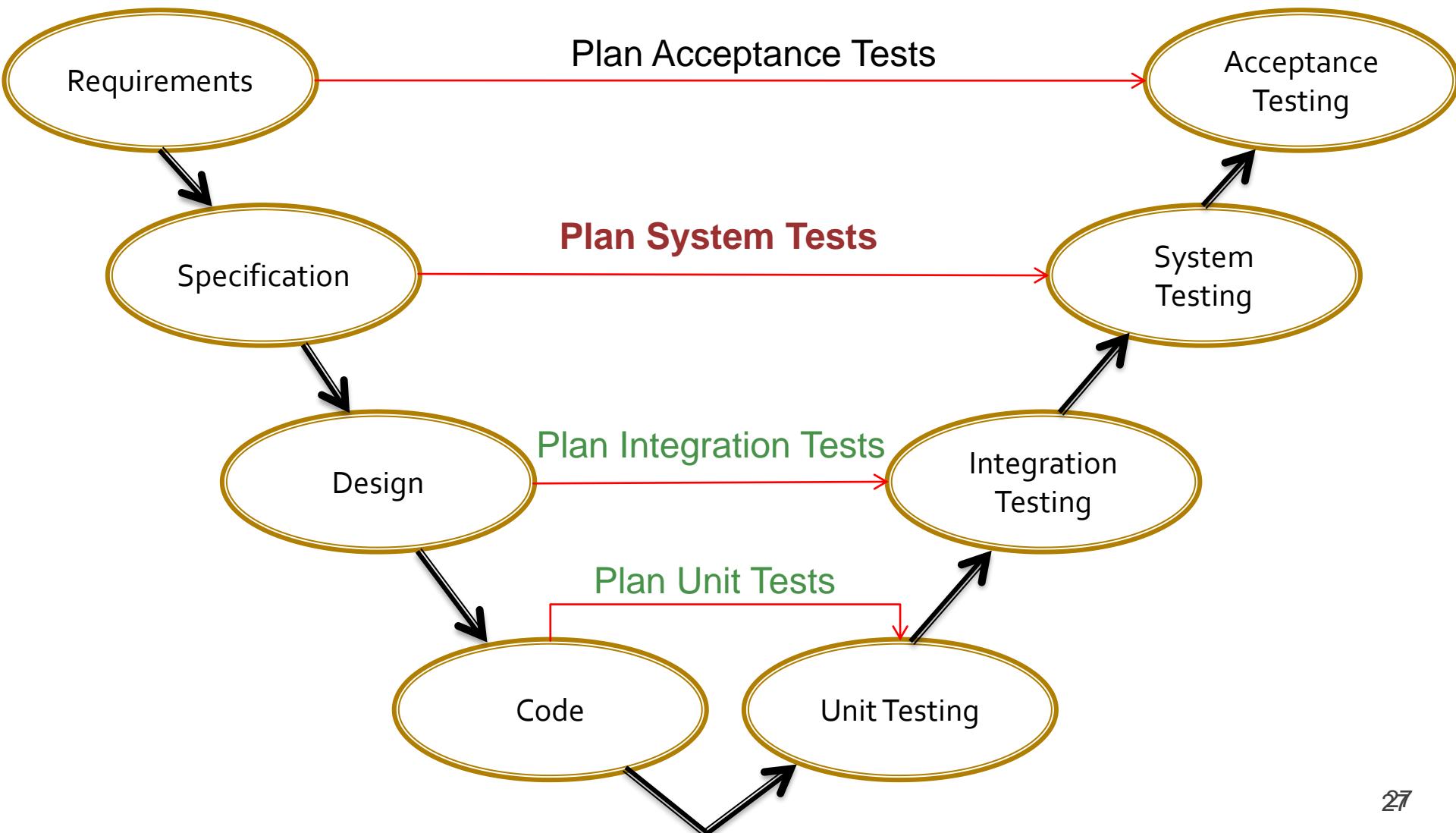
- It occurs after **unit testing** and before **system testing**
  - Takes as its input **modules** that have been unit tested
  - Groups them in larger aggregates
  - Applies tests defined in an **integration test plan** to those aggregates
  - Delivers as its output the integrated system ready for system testing.
- Purpose
  - Testing functionality, reliability, and performance of Major Design Components
    - E.g. separate subsystems of overall system
- Major approaches
  - **Top-down testing**
    - The top integrated modules are tested and the **branch** of the module is tested step by step until the end of the related module.
  - **Bottom-up testing**
    - The lowest level components are tested first, then used to facilitate the testing of higher level components

# Sample Integration Test Plan

[Link 1](#)

[Link](#)

# Testing in V-Model



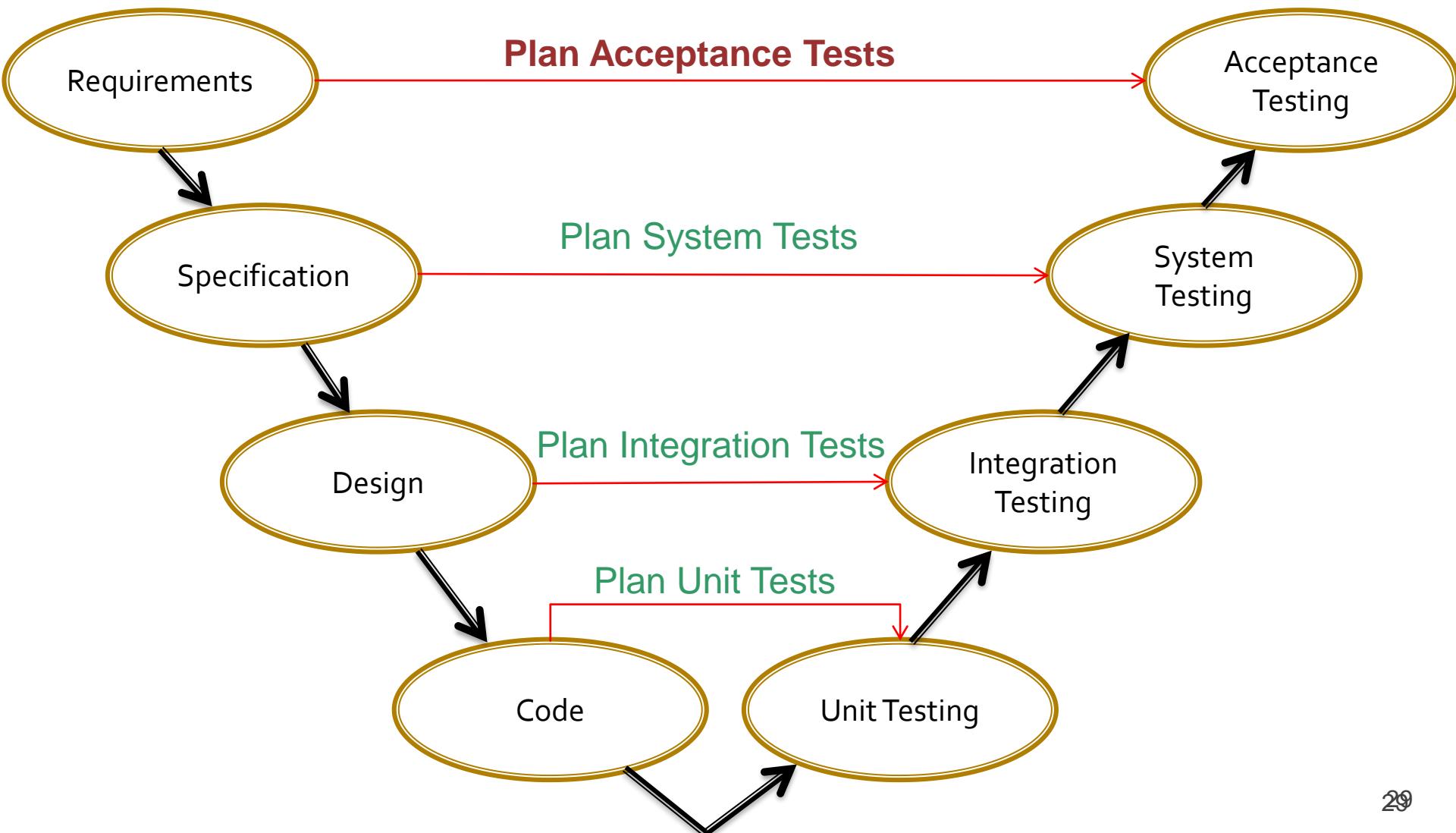
# System Testing

*Testing a complete, integrated system to verify it matches its requirements (user requirements and system requirements)*

- Tests that can be performed in System Testing:

- GUI software testing
- Usability testing
- Performance testing
- Compatibility testing
- Error handling testing
- Load testing
- Volume testing
- Stress testing
- Security testing
- Scalability testing
- Sanity testing
- Smoke testing
- Exploratory testing
- Ad hoc testing
- Regression testing
- Reliability testing
- Installation testing
- Maintenance testing
- Recovery testing

# Testing in V-Model



# Acceptance Testing

*A process to obtain **confirmation** that a system meets mutually agreed-upon requirements*

- Acceptance Tests/Criteria are usually **created by business customers** and expressed in a **business domain language**.
  - High-level tests to test the completeness of a user story during any sprint
- Acceptance phase may also act as the final quality gateway
- Acceptance testing performed by the customer is also known as
  - user acceptance testing (UAT)
  - end-user testing
  - site (acceptance) testing
  - field (acceptance) testing.

# Approaches for acceptance testing

- **Manual Acceptance testing.** User exercises the system manually using his creativity.
- **Acceptance testing with “GUI Test Drivers”** (at the GUI level). These tools help the developer do functional/acceptance testing through a user interface such as a native GUI or web interface. **“Capture and Replay” Tools capture events** (e.g. mouse, keyboard) in modifiable script.

**Disadvantages:**  
expensive, error prone,  
not repeatable, ...

**Disadvantages:**  
Tests are brittle, i.e., have  
to be re-captured if the  
GUI changes.

***“Avoid acceptance testing only in final stage: Too late to find bugs”***

# Agile and Testing

- Agile team structure?
- In Agile Development it might *not* be needed to have a separate team of testers
  - Not having a separate test team does not mean that Agile considers testing to be any less important
- Basics of Agile Testing:
  - Testing can done **more effectively** in **short turn around times**, by people who know the system and its objectives very well
    - Developers write **unit tests** as part of their coding activity
    - Business analyst or Product owner to define and in some cases run the **acceptance tests**
  - **Automated Testing**
    - Can be simple test-running scripts

# Test-Driven Development (TDD)

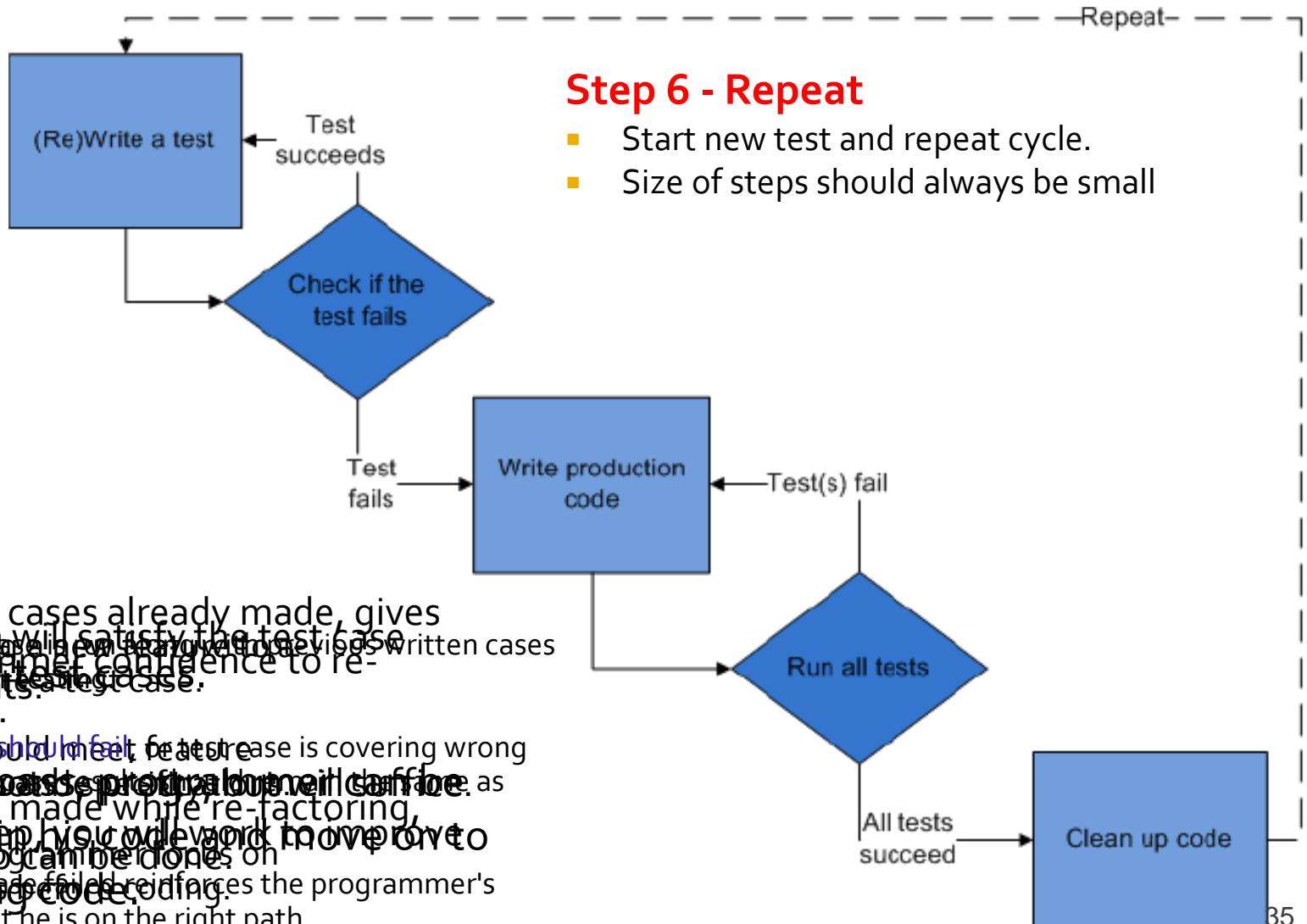
*TDD is a **software development process** that relies on the repetition of a very short development cycle.*

- Related to the **test-first programming** concepts of eXtreme Programming in 1999.
- Requires developers to create automated unit tests that define code requirements **BEFORE** writing the code itself.
- This type of method has been experienced in CSC108 and other previous programming courses.

# Test-Driven Development Cycle

1. Add a New Test
2. Run Tests and Verify New Test
3. Write Code to Satisfy New Test
4. Run Tests and Check Validity
5. Re-factor Code
6. Repeat

# Test-Driven Development Cycle



# TDD Process

## 1. Add a New Test

- Before adding a new feature to a program, write a test case.
- Test case should meet feature requirements and specifications.
- Helps the programmer focus on requirements before coding.

## 2. Run Tests and Verify New Test

- The new test case is run along with previous written cases to verify proper testing.
- New test case should fail, or test case is covering wrong feature. Other case results should remain the same as before.
- Knowing the case failed reinforces the programmer's confidence that he is on the right path.

## 3. Write code

- Code which will satisfy the test case requirements.
- Code may not be pretty, but will suffice. In a later step, you will work to improve the code.

# TDD Process

## 4. Run Tests

- Run all the test cases.
- If all tests pass, programmer can be confident in his code and move on to re-factoring code.

## 5. Re-factor Code

- Having test cases already made, gives the programmer confidence to re-factor code.
- If errors are made while re-factoring, simple undo can be done.

## 6. Repeat

- Start new test and repeat cycle.
- Size of steps should always be small

# TDD

TDD is a *confidence builder* and helps programmers focus on the requirements. When a test fails, progress has been made since the programmer knows the correct problem has to be solved. There is a clear **measure of success** after passing a test. By showing that the function works and meets the requirement, it gives the developer **confidence to move on**. The point of TDD is to "**test with a purpose**".

## Disadvantages:

- False sense of security
  - All of the tests that developers had written were passed, but they neglected some of the important faults !!
- Excessive number of test cases, reduced productivity

# Chapter 29

---

## ■ Software Configuration Management

*Slide Set to accompany*

*Software Engineering: A Practitioner's Approach, 8/e*  
**by Roger S. Pressman and Bruce R. Maxim**

Slides copyright © 1996, 2001, 2005, 2009, 2014 by Roger S. Pressman

***For non-profit educational use only***

May be reproduced ONLY for student use at the university level when used in conjunction with *Software Engineering: A Practitioner's Approach, 8/e*. Any other reproduction or use is prohibited without the express written permission of the author.

All copyright information MUST appear if these slides are posted on a website for student use.

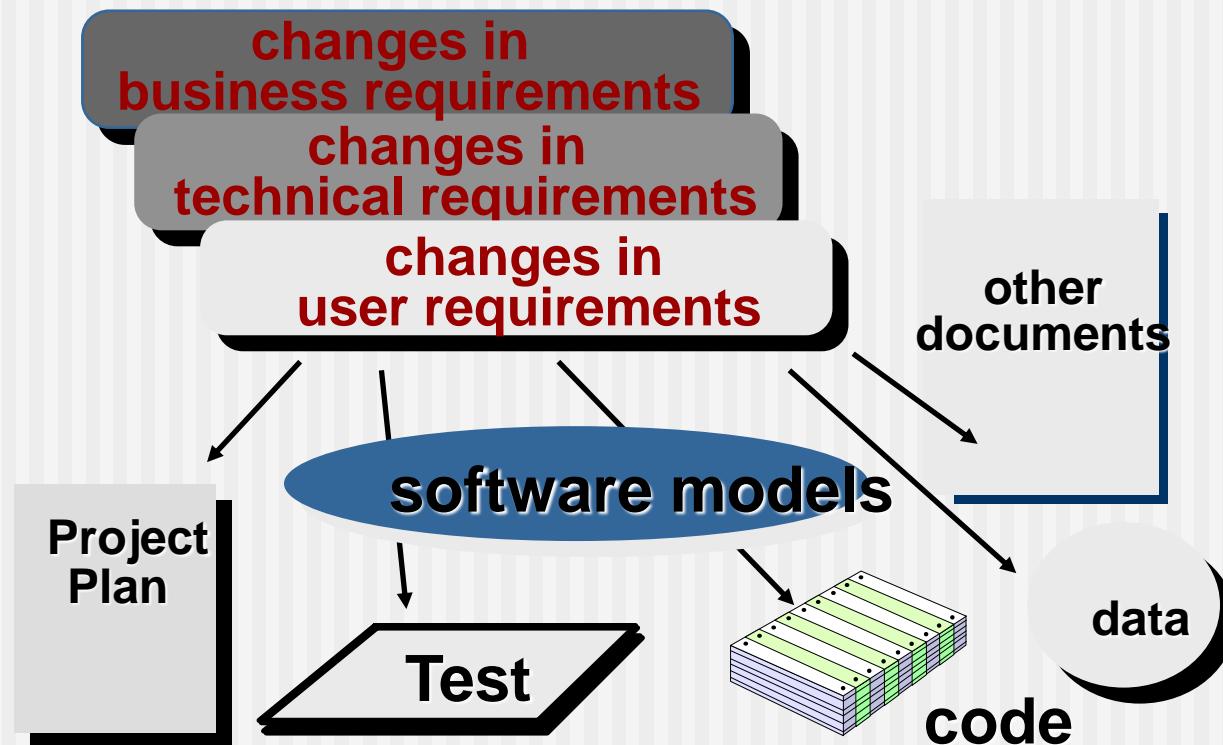
# The “First Law”

---

**No matter where you are in the system life cycle, the system will change, and the desire to change it will persist throughout the life cycle.**

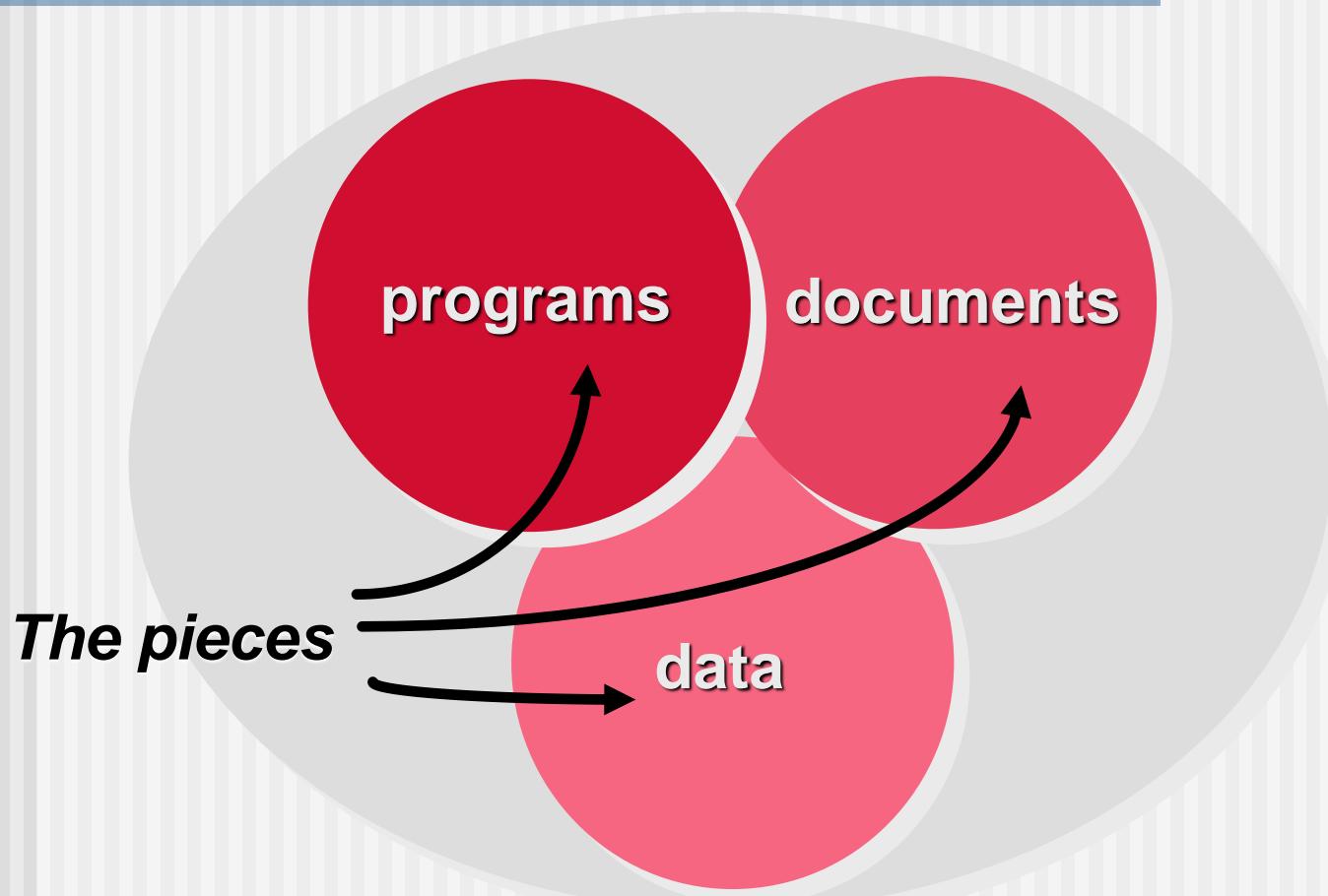
*Bersoff, et al, 1980*

# What Are These Changes?



# The Software Configuration

---

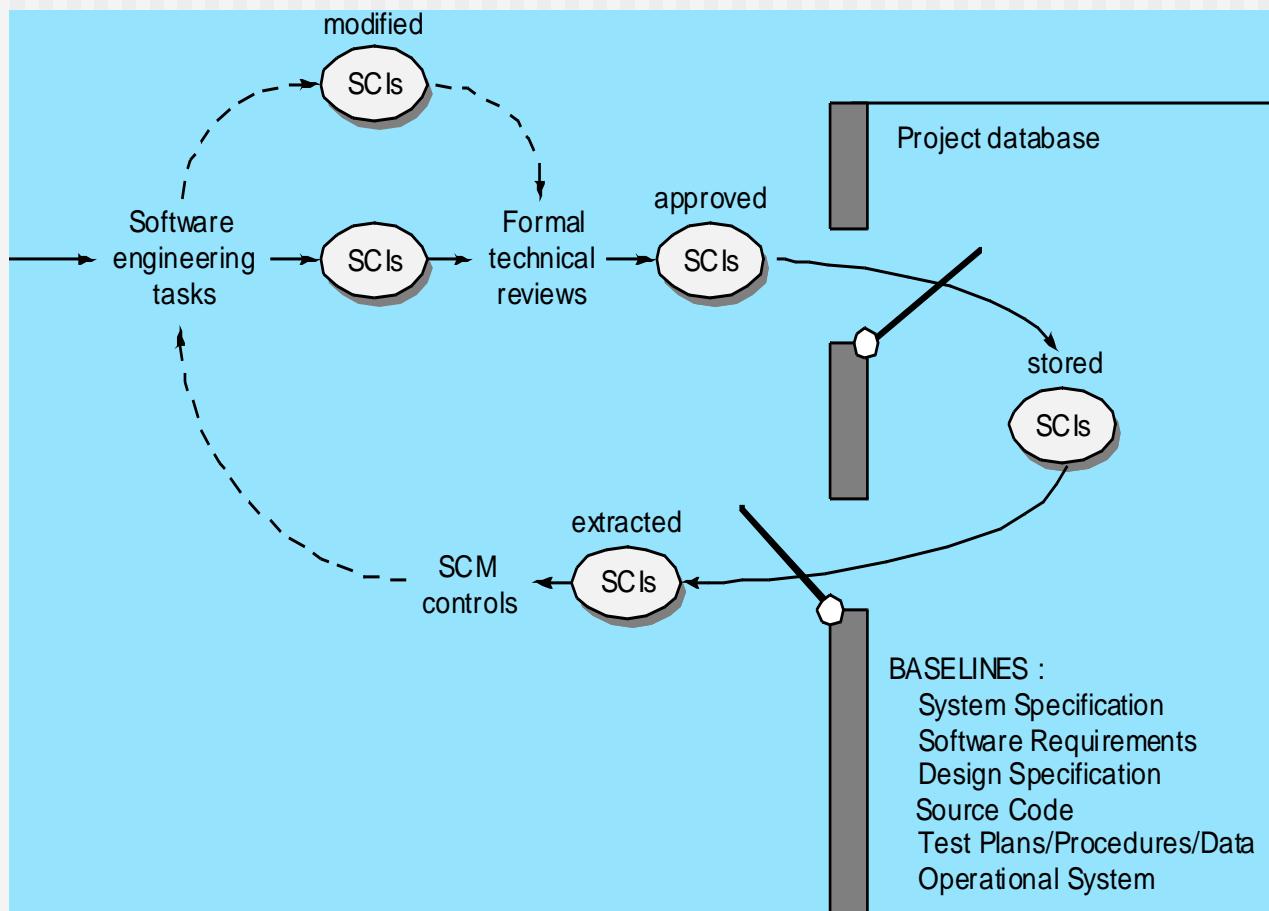


# Baselines

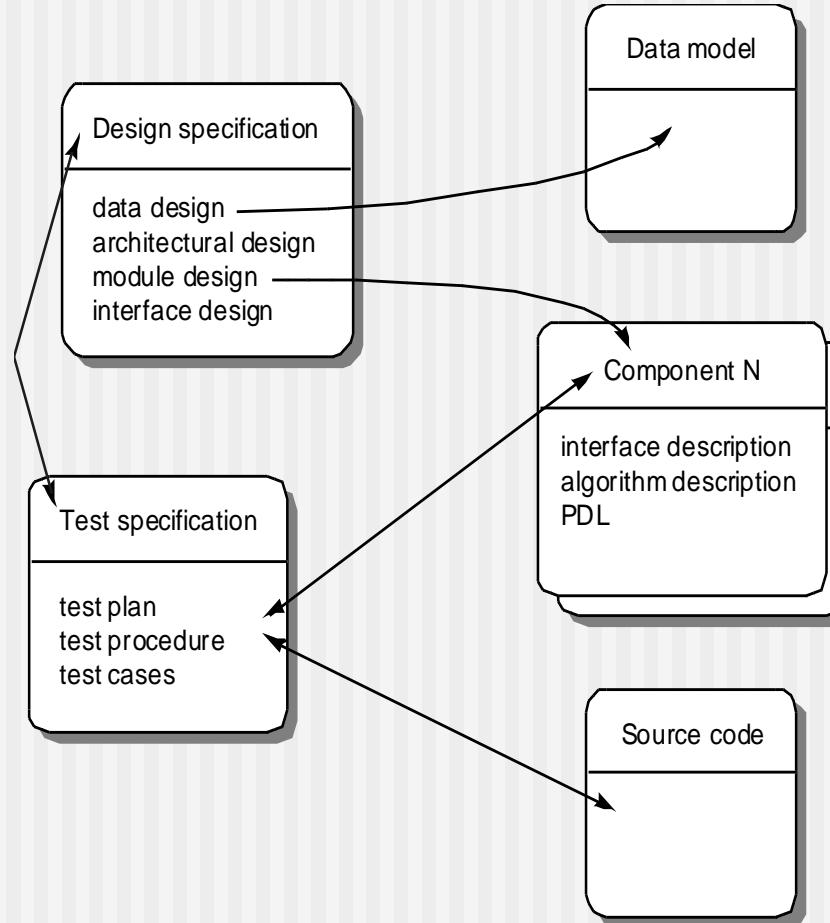
---

- The IEEE (IEEE Std. No. 610.12-1990) defines a baseline as:
  - A specification or product that has been formally reviewed and agreed upon, that thereafter serves as the basis for further development, and that can be changed only through formal change control procedures.
- a baseline is a milestone in the development of software that is marked by the delivery of one or more software configuration items and the approval of these SCIs that is obtained through a formal technical review

# Baselines



# Software Configuration Objects

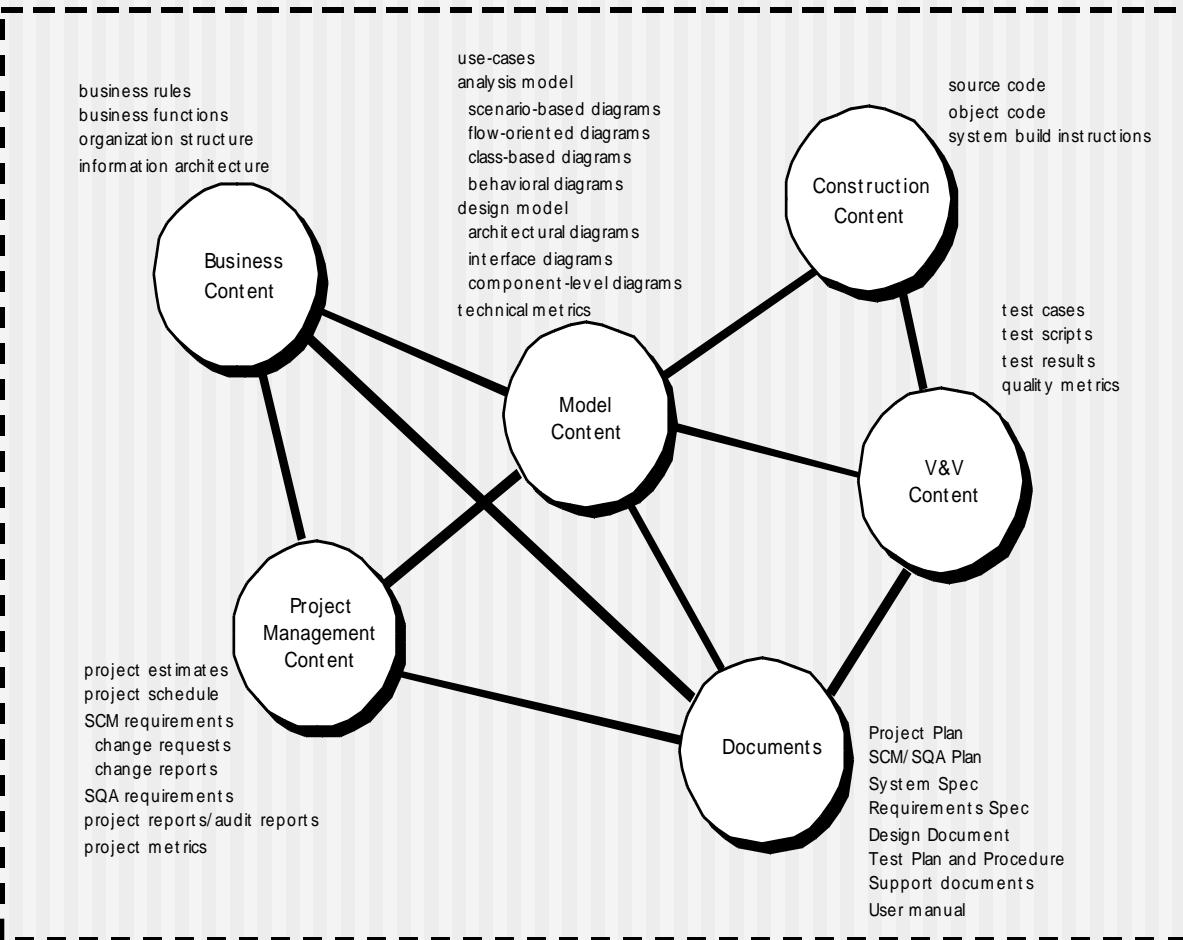


# SCM Repository

---

- The SCM repository is the set of mechanisms and data structures that allow a software team to manage change in an effective manner
- The repository performs or precipitates the following functions [For89]:
  - Data integrity
  - Information sharing
  - Tool integration
  - Data integration
  - Methodology enforcement
  - Document standardization

# Repository Content



# Repository Features

---

- **Versioning.**
  - saves all of these versions to enable effective management of product releases and to permit developers to go back to previous versions
- **Dependency tracking and change management.**
  - The repository manages a wide variety of relationships among the data elements stored in it.
- **Requirements tracing.**
  - Provides the ability to track all the design and construction components and deliverables that result from a specific requirement specification
- **Configuration management.**
  - Keeps track of a series of configurations representing specific project milestones or production releases. Version management provides the needed versions, and link management keeps track of interdependencies.
- **Audit trails.**
  - establishes additional information about when, why, and by whom changes are made.

# SCM Elements

---

- *Component elements*—a set of tools coupled within a file management system (e.g., a database) that enables access to and management of each software configuration item.
- *Process elements*—a collection of procedures and tasks that define an effective approach to change management (and related activities) for all constituencies involved in the management, engineering and use of computer software.
- *Construction elements*—a set of tools that automate the construction of software by ensuring that the proper set of validated components (i.e., the correct version) have been assembled.
- *Human elements*—to implement effective SCM, the software team uses a set of tools and process features (encompassing other CM elements)

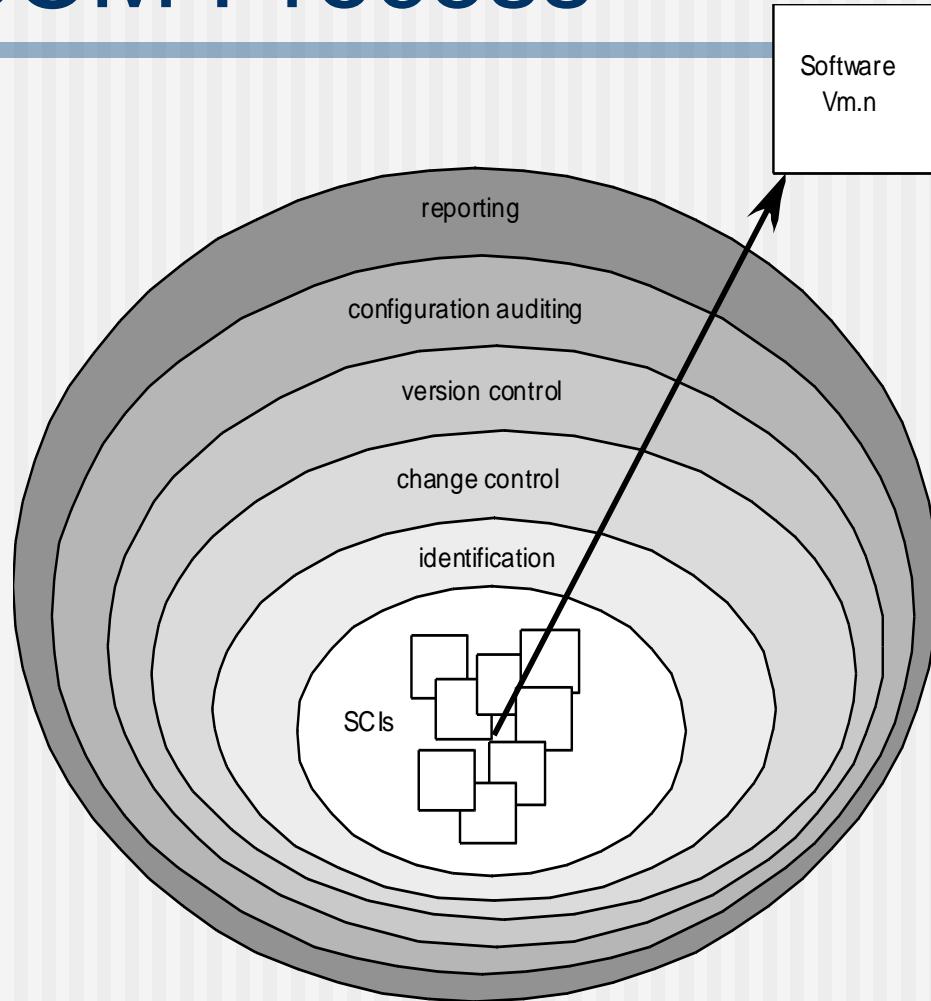
# The SCM Process

---

*Addresses the following questions ...*

- How does a software team identify the discrete elements of a software configuration?
- How does an organization manage the many existing versions of a program (and its documentation) in a manner that will enable change to be accommodated efficiently?
- How does an organization control changes before and after software is released to a customer?
- Who has responsibility for approving and ranking changes?
- How can we ensure that changes have been made properly?
- What mechanism is used to appraise others of changes that are made?

# The SCM Process



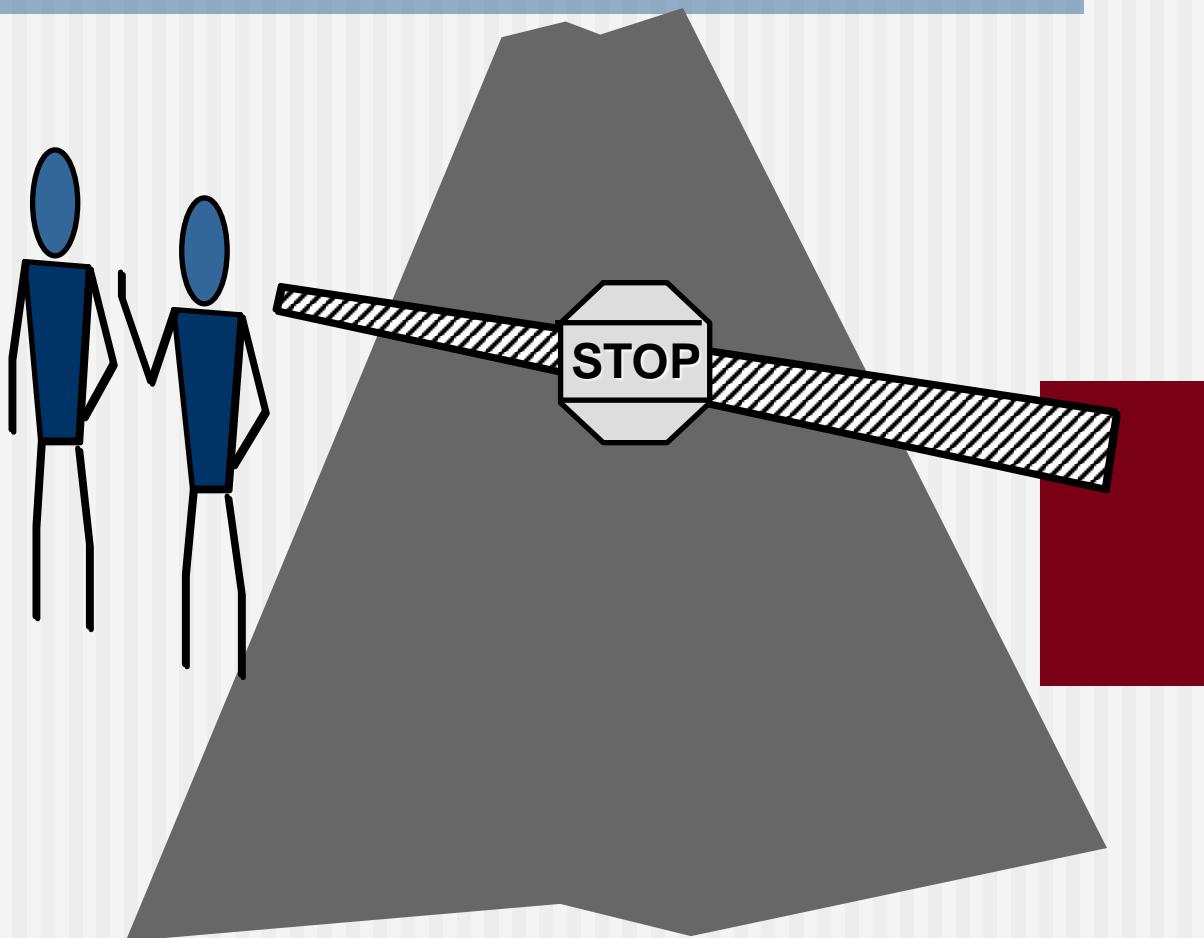
# Version Control

---

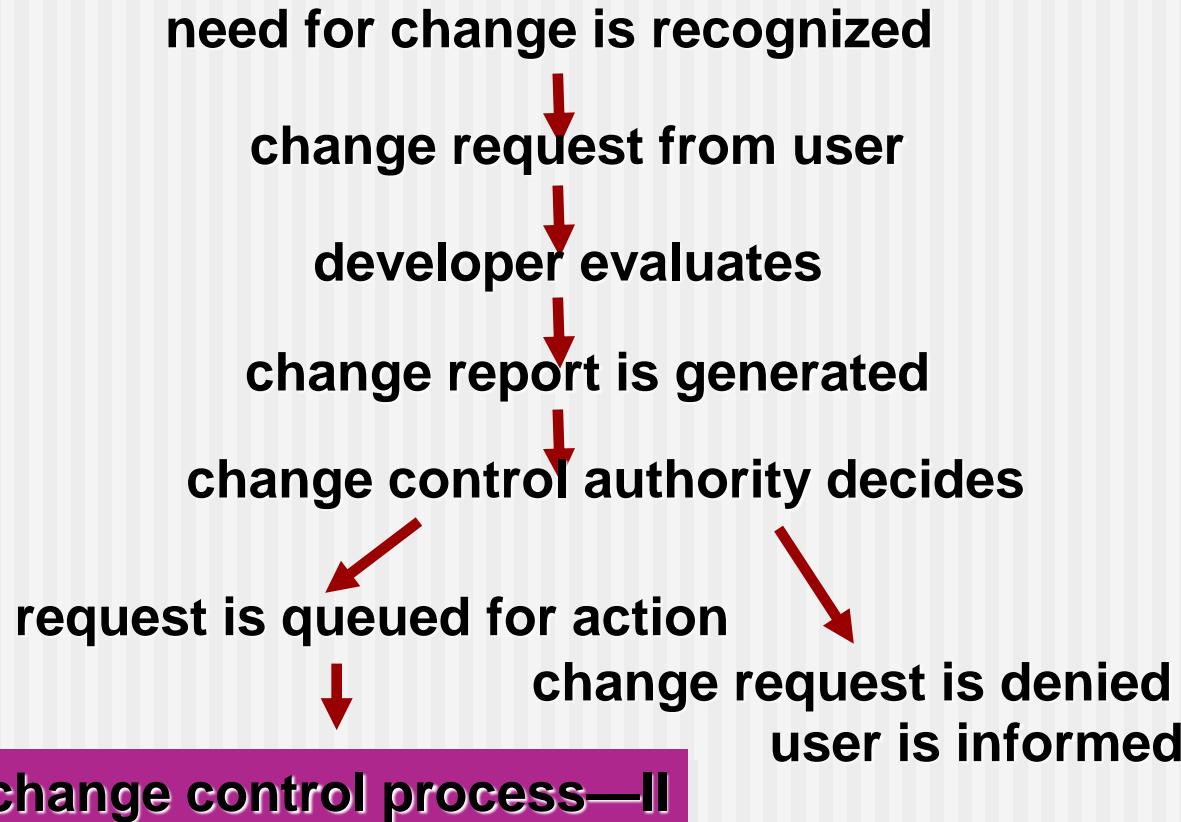
- Version control combines procedures and tools to manage different versions of configuration objects that are created during the software process
- A version control system implements or is directly integrated with four major capabilities:
  - a *project database (repository)* that stores all relevant configuration objects
  - a *version management* capability that stores all versions of a configuration object (or enables any version to be constructed using differences from past versions);
  - a *make facility* that enables the software engineer to collect all relevant configuration objects and construct a specific version of the software.
  - an *issues tracking* (also called *bug tracking*) capability that enables the team to record and track the status of all outstanding issues associated with each configuration object.

# Change Control

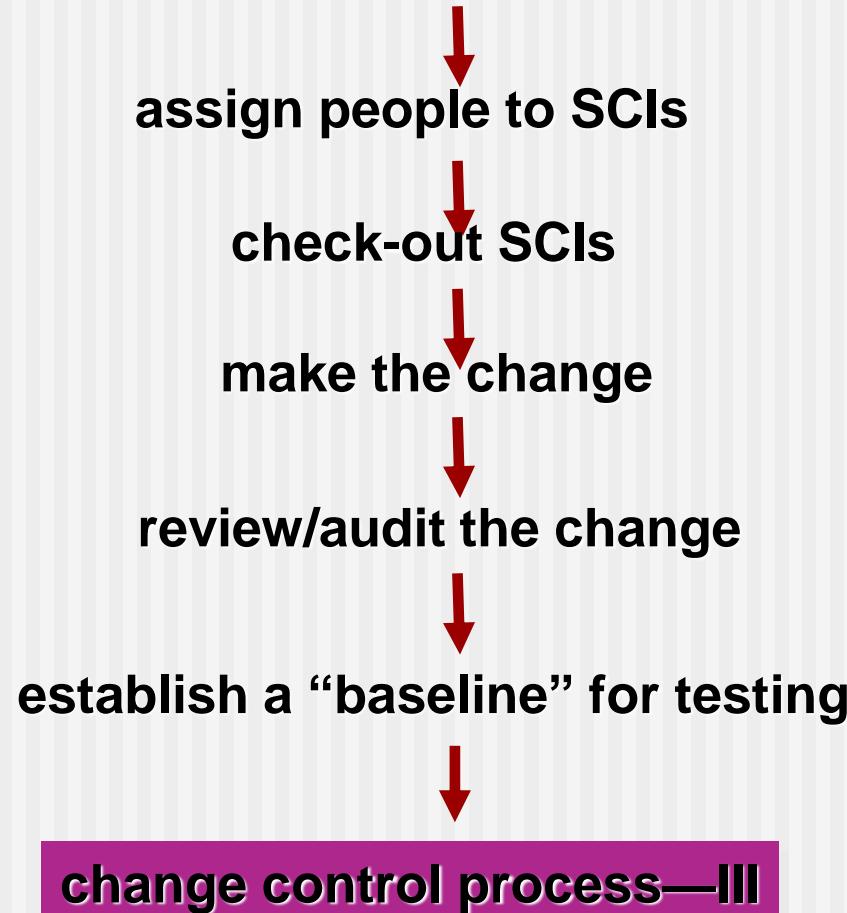
---



# Change Control Process—I

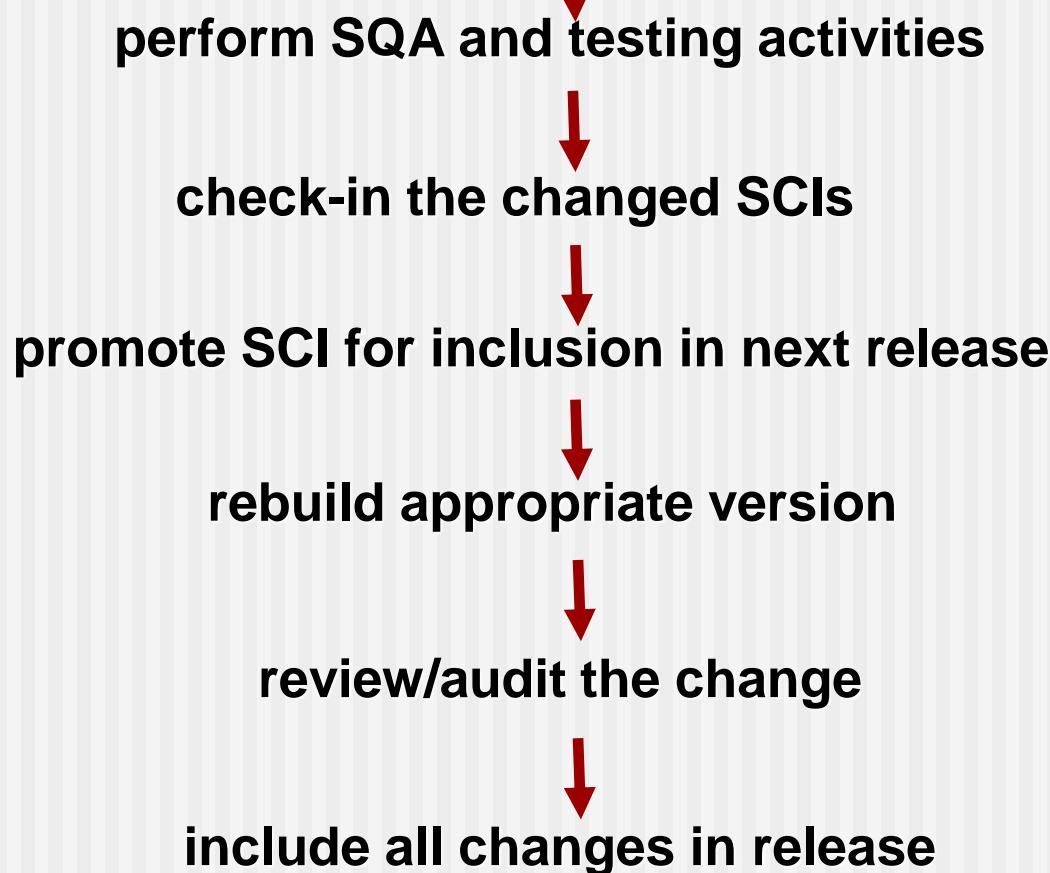


# Change Control Process-II



# Change Control Process-III

---



# Chapter 21

---

## ■ Software Quality Assurance

*Slide Set to accompany*

*Software Engineering: A Practitioner's Approach, 8/e*  
**by Roger S. Pressman and Bruce R. Maxim**

Slides copyright © 1996, 2001, 2005, 2099, 2014 by Roger S. Pressman

***For non-profit educational use only***

May be reproduced ONLY for student use at the university level when used in conjunction with *Software Engineering: A Practitioner's Approach, 8/e*. Any other reproduction or use is prohibited without the express written permission of the author.

All copyright information MUST appear if these slides are posted on a website for student use.

# Elements of SQA

---

- **Standards**
- **Reviews and Audits**
- **Testing**
- **Error/defect collection and analysis**
- **Change management**
- **Education**
- **Vendor management**
- **Security management**
- **Safety**
- **Risk management**

# Role of the SQA Group-I

---

- **Prepares an SQA plan for a project.**
  - The plan identifies
    - evaluations to be performed
    - audits and reviews to be performed
    - standards that are applicable to the project
    - procedures for error reporting and tracking
    - documents to be produced by the SQA group
    - amount of feedback provided to the software project team
- **Participates in the development of the project's software process description.**
  - The SQA group reviews the process description for compliance with organizational policy, internal software standards, externally imposed standards (e.g., ISO-9001), and other parts of the software project plan.

# Role of the SQA Group-II

---

- **Reviews software engineering activities to verify compliance with the defined software process.**
  - identifies, documents, and tracks deviations from the process and verifies that corrections have been made.
- **Audits designated software work products to verify compliance with those defined as part of the software process.**
  - reviews selected work products; identifies, documents, and tracks deviations; verifies that corrections have been made
  - periodically reports the results of its work to the project manager.
- **Ensures that deviations in software work and work products are documented and handled according to a documented procedure.**
- **Records any noncompliance and reports to senior management.**
  - Noncompliance items are tracked until they are resolved.

# SQA Goals (see Figure 21.1)

---

- **Requirements quality.** The correctness, completeness, and consistency of the requirements model will have a strong influence on the quality of all work products that follow.
- **Design quality.** Every element of the design model should be assessed by the software team to ensure that it exhibits high quality and that the design itself conforms to requirements.
- **Code quality.** Source code and related work products (e.g., other descriptive information) must conform to local coding standards and exhibit characteristics that will facilitate maintainability.
- **Quality control effectiveness.** A software team should apply limited resources in a way that has the highest likelihood of achieving a high quality result.

# Formal SQA

---

- Assumes that a rigorous syntax and semantics can be defined for every programming language
- Allows the use of a rigorous approach to the specification of software requirements
- Applies mathematical proof of correctness techniques to demonstrate that a program conforms to its specification

# Statistical SQA

---

- Information about software errors and defects is collected and categorized.
- An attempt is made to trace each error and defect to its underlying cause (e.g., non-conformance to specifications, design error, violation of standards, poor communication with the customer).
- Using the Pareto principle (80 percent of the defects can be traced to 20 percent of all possible causes), isolate the 20 percent (the *vital few*).
- Once the vital few causes have been identified, move to correct the problems that have caused the errors and defects.

# Software Reliability

---

- A simple measure of reliability is *mean-time-between-failure* (MTBF), where

$$\text{MTBF} = \text{MTTF} + \text{MTTR}$$

- The acronyms MTTF and MTTR are *mean-time-to-failure* and *mean-time-to-repair*, respectively.
- *Software availability* is the probability that a program is operating according to requirements at a given point in time and is defined as

$$\text{Availability} = [\text{MTTF}/(\text{MTTF} + \text{MTTR})] \times 100\%$$

# Software Safety

---

- *Software safety* is a software quality assurance activity that focuses on the identification and assessment of potential hazards that may affect software negatively and cause an entire system to fail.
- If hazards can be identified early in the software process, software design features can be specified that will either eliminate or control potential hazards.

# ISO 9001:2008 Standard

---

- ISO 9001:2008 is the quality assurance standard that applies to software engineering.
- The standard contains 20 requirements that must be present for an effective quality assurance system.
- The requirements delineated by ISO 9001:2008 address topics such as
  - management responsibility, quality system, contract review, design control, document and data control, product identification and traceability, process control, inspection and testing, corrective and preventive action, control of quality records, internal quality audits, training, servicing, and statistical techniques.

# Chapter 20

---

## ■ Review Techniques

*Slide Set to accompany*

*Software Engineering: A Practitioner's Approach, 8/e*  
**by Roger S. Pressman and Bruce R. Maxim**

Slides copyright © 1996, 2001, 2005, 2009, 2014 by Roger S. Pressman

***For non-profit educational use only***

May be reproduced ONLY for student use at the university level when used in conjunction with *Software Engineering: A Practitioner's Approach, 8/e*. Any other reproduction or use is prohibited without the express written permission of the author.

All copyright information MUST appear if these slides are posted on a website for student use.

# What Are Reviews?

---

- a meeting conducted by technical people for technical people
- a technical assessment of a work product created during the software engineering process
- a software quality assurance mechanism
- a training ground

# What Reviews Are Not

---

- A project summary or progress assessment
- A meeting intended solely to impart information
- A mechanism for political or personal reprisal!

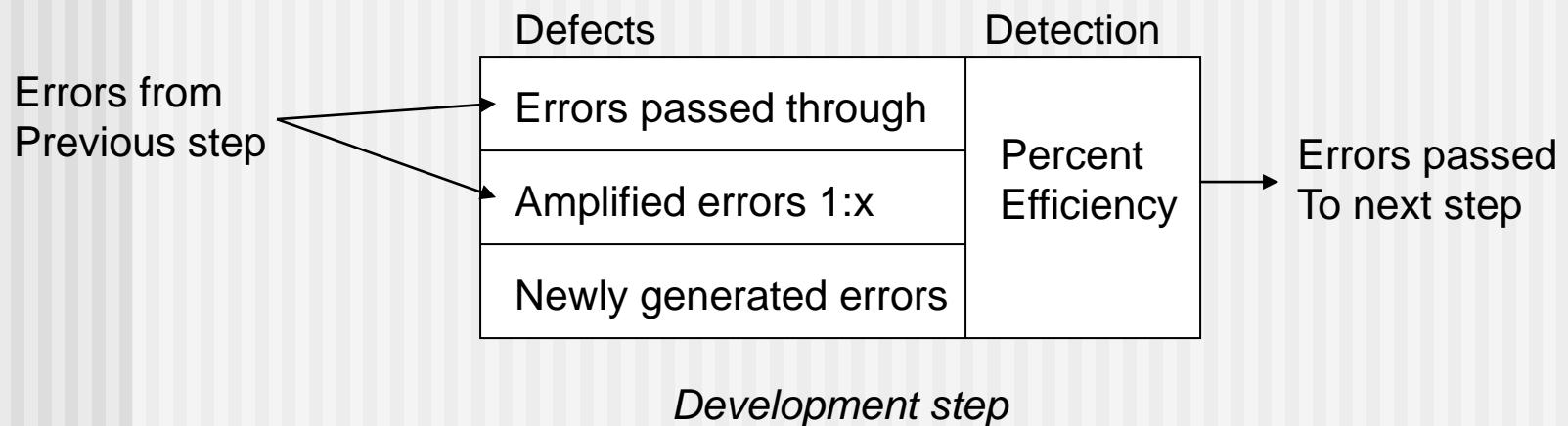
# What Do We Look For?

---

- Errors and defects
  - *Error*—a quality problem found *before* the software is released to end users
  - *Defect*—a quality problem found only *after* the software has been released to end-users
- We make this distinction because errors and defects have very different economic, business, psychological, and human impact
- However, the temporal distinction made between errors and defects in this book is *not* mainstream thinking

# Defect Amplification

- A *defect amplification model* [IBM81] can be used to illustrate the generation and detection of errors during the design and code generation actions of a software process.



# Defect Amplification

---

- In the example provided in SEPA, Section 20.2,
  - a software process that does NOT include reviews,
    - yields **94 errors** at the beginning of testing and
    - Releases **12 latent defects** to the field
  - a software process that does include reviews,
    - yields **24 errors** at the beginning of testing and
    - releases **3 latent defects** to the field
  - A cost analysis indicates that the process with NO reviews costs **approximately 3 times** more than the process with reviews, taking the cost of correcting the latent defects into account

# Metrics

---

- The total review effort and the total number of errors discovered are defined as:
  - $E_{review} = E_p + E_a + E_r$
  - $Err_{tot} = Err_{minor} + Err_{major}$
- *Defect density* represents the errors found per unit of work product reviewed.
  - Defect density =  $Err_{tot} / WPS$
- where ...

# Metrics

---

- *Preparation effort,  $E_p$*  — the effort (in person-hours) required to review a work product prior to the actual review meeting
- *Assessment effort,  $E_a$*  — the effort (in person-hours) that is expending during the actual review
- *Rework effort,  $E_r$*  — the effort (in person-hours) that is dedicated to the correction of those errors uncovered during the review
- *Work product size, WPS* — a measure of the size of the work product that has been reviewed (e.g., the number of UML models, or the number of document pages, or the number of lines of code)
- *Minor errors found,  $Err_{minor}$*  — the number of errors found that can be categorized as minor (requiring less than some pre-specified effort to correct)
- *Major errors found,  $Err_{major}$*  — the number of errors found that can be categorized as major (requiring more than some pre-specified effort to correct)

# Informal Reviews

---

- Informal reviews include:
  - a simple desk check of a software engineering work product with a colleague
  - a casual meeting (involving more than 2 people) for the purpose of reviewing a work product, or
  - the review-oriented aspects of pair programming
- *pair programming* encourages continuous review as a work product (design or code) is created.
  - The benefit is immediate discovery of errors and better work product quality as a consequence.

# Formal Technical Reviews

---

- The objectives of an FTR are:
  - to uncover errors in function, logic, or implementation for any representation of the software
  - to verify that the software under review meets its requirements
  - to ensure that the software has been represented according to predefined standards
  - to achieve software that is developed in a uniform manner
  - to make projects more manageable
- The FTR is actually a class of reviews that includes *walkthroughs* and *inspections*.

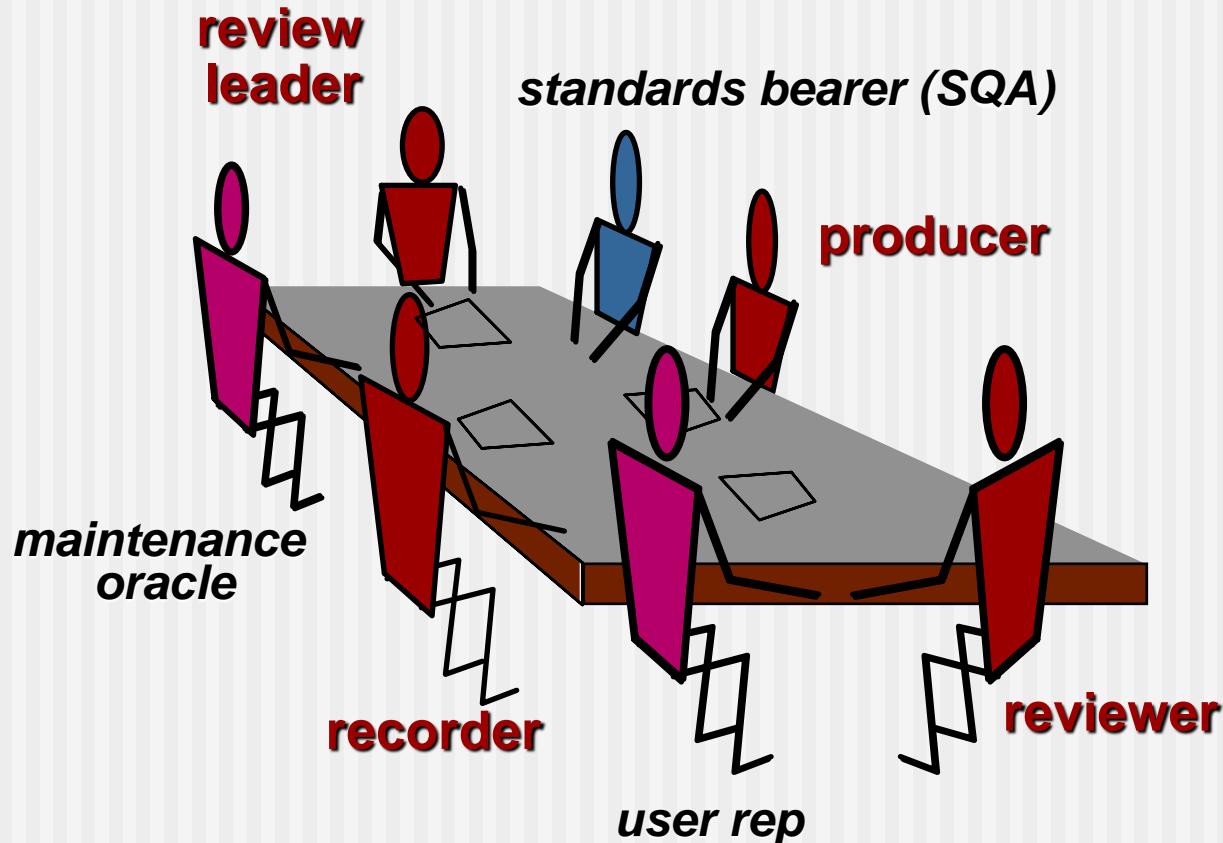
# The Review Meeting

---

- Between three and five people (typically) should be involved in the review.
- Advance preparation should occur but should require no more than two hours of work for each person.
- The duration of the review meeting should be less than two hours.
- Focus is on a work product (e.g., a portion of a requirements model, a detailed component design, source code for a component)

# The Players

---



# The Players

---

- *Producer*—the individual who has developed the work product
  - informs the project leader that the work product is complete and that a review is required
- *Review leader*—evaluates the product for readiness, generates copies of product materials, and distributes them to two or three *reviewers* for advance preparation.
- *Reviewer(s)*—expected to spend between one and two hours reviewing the product, making notes, and otherwise becoming familiar with the work.
- *Recorder*—reviewer who records (in writing) all important issues raised during the review.

# Metrics Derived from Reviews

---

- inspection time per page of documentation
- inspection time per KLOC or FP
- inspection effort per KLOC or FP
- errors uncovered per reviewer hour
- errors uncovered per preparation hour
- errors uncovered per SE task (e.g., design)
- number of minor errors (e.g., typos)
- number of major errors  
(e.g., nonconformance to req.)
- number of errors found during preparation

# Chapter 33

---

## ■ Estimation for Software Projects

*Slide Set to accompany*

*Software Engineering: A Practitioner's Approach, 8/e*  
**by Roger S. Pressman and Bruce R. Maxim**

Slides copyright © 1996, 2001, 2005, 2009, 2014 by Roger S. Pressman

***For non-profit educational use only***

May be reproduced ONLY for student use at the university level when used in conjunction with *Software Engineering: A Practitioner's Approach, 8/e*. Any other reproduction or use is prohibited without the express written permission of the author.

All copyright information MUST appear if these slides are posted on a website for student use.

# Software Project Planning

The overall goal of project planning is to establish a pragmatic strategy for controlling, tracking, and monitoring a complex technical project.

Why?

***So the end result gets done on time, with quality!***

# Project Planning Task Set-I

---

- Establish project scope
- Determine feasibility
- Analyze risks
  - Risk analysis is considered in detail in Chapter 25.
- Define required resources
  - Determine require human resources
  - Define reusable software resources
  - Identify environmental resources

# Project Planning Task Set-II

---

- Estimate cost and effort
  - Decompose the problem
  - Develop two or more estimates using size, function points, process tasks or use-cases
  - Reconcile the estimates
- Develop a project schedule
  - Scheduling is considered in detail in Chapter 34.
    - Establish a meaningful task set
    - Define a task network
    - Use scheduling tools to develop a timeline chart
    - Define schedule tracking mechanisms

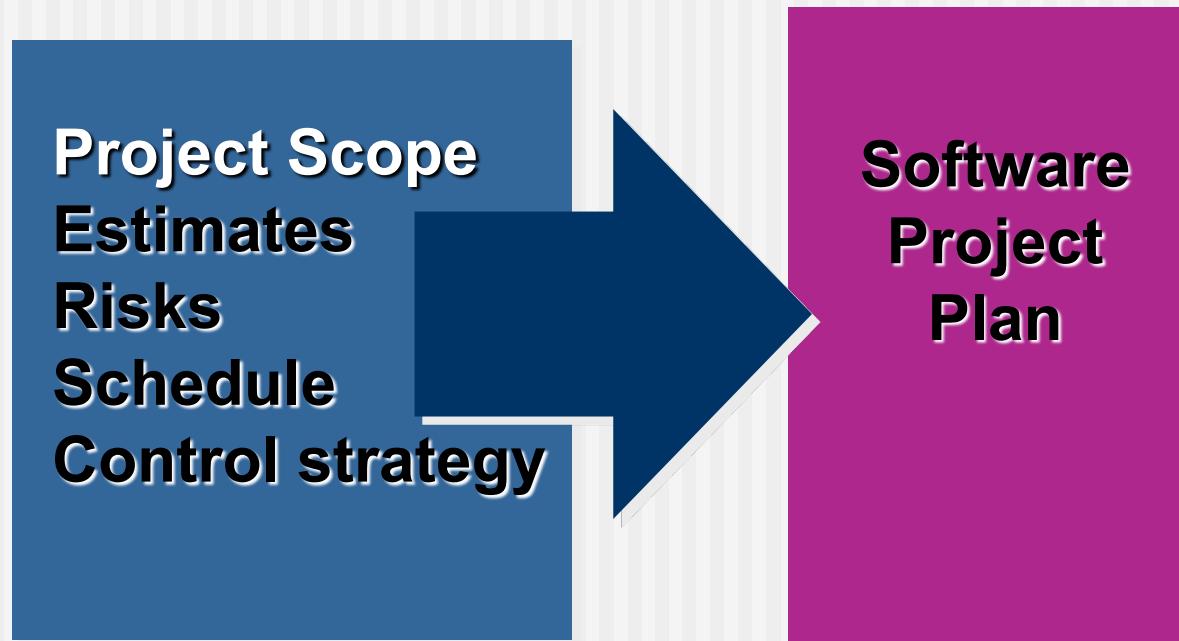
# Estimation

---

- Estimation of resources, cost, and schedule for a software engineering effort requires
  - experience
  - access to good historical information (metrics)
  - the courage to commit to quantitative predictions when qualitative information is all that exists
- Estimation carries inherent risk and this risk leads to uncertainty

# Write it Down!

---



# To Understand Scope ...

---

- Understand the customers needs
- understand the business context
- understand the project boundaries
- understand the customer's motivation
- understand the likely paths for change
- understand that ...

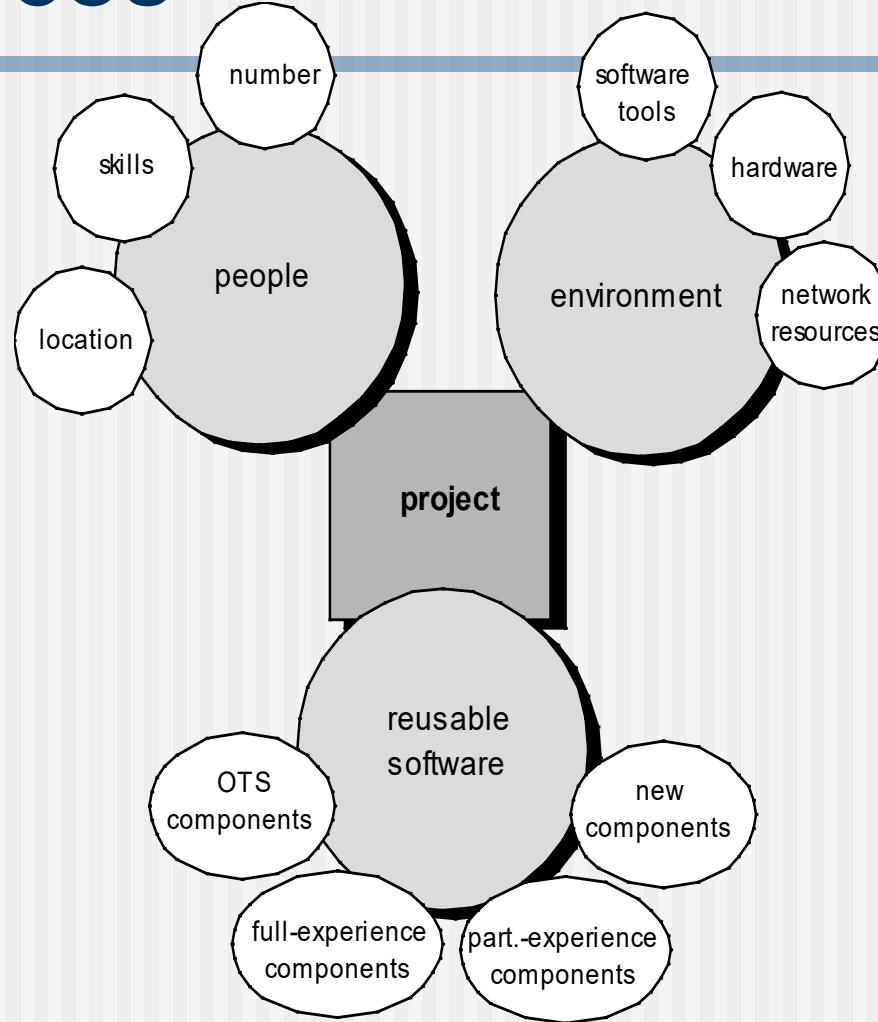
***Even when you understand,  
nothing is guaranteed!***

# What is Scope?

---

- *Software scope* describes
  - the functions and features that are to be delivered to end-users
  - the data that are input and output
  - the “content” that is presented to users as a consequence of using the software
  - the performance, constraints, interfaces, and reliability that *bound* the system.
- Scope is defined using one of two techniques:
  - A narrative description of software scope is developed after communication with all stakeholders.
  - A set of use-cases is developed by end-users.

# Resources



# Project Estimation

---



- Project scope must be understood
- Elaboration (decomposition) is necessary
- Historical metrics are very helpful
- At least two different techniques should be used
- Uncertainty is inherent in the process

# Estimation Techniques

---

- Past (similar) project experience
- Conventional estimation techniques
  - task breakdown and effort estimates
  - size (e.g., FP) estimates
- Empirical models
- Automated tools

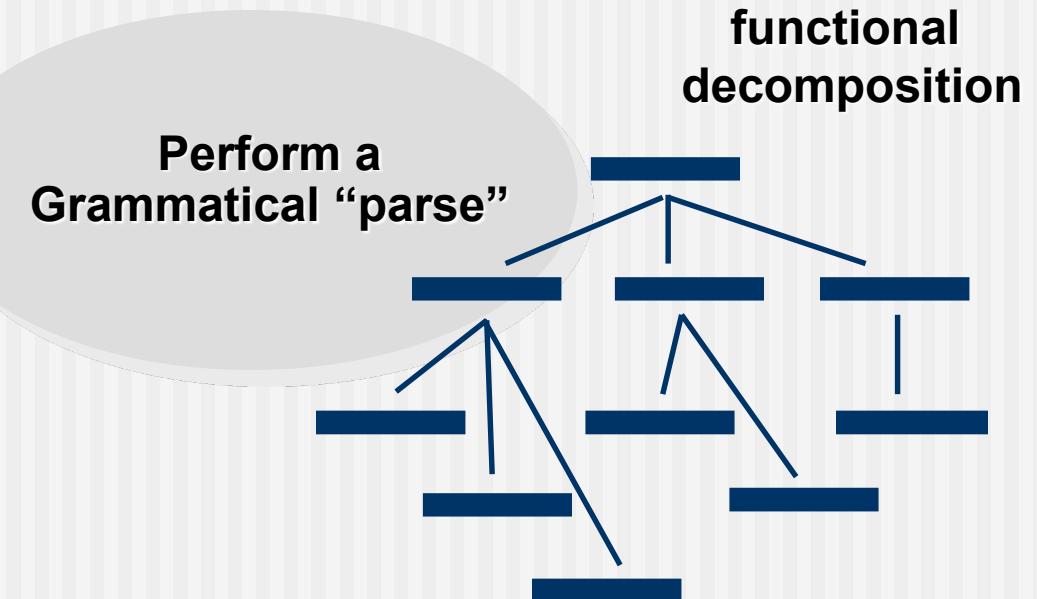
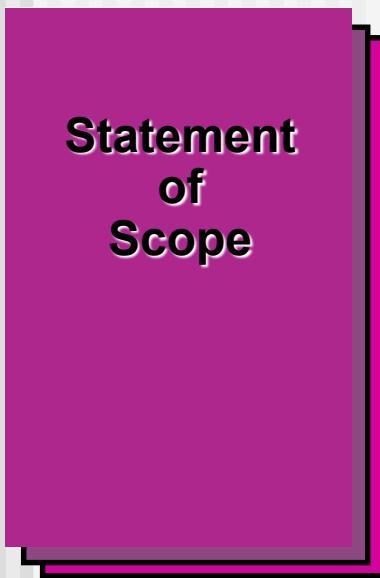


# Estimation Accuracy

---

- Predicated on ...
  - the degree to which the planner has properly estimated the size of the product to be built
  - the ability to translate the size estimate into human effort, calendar time, and dollars (a function of the availability of reliable software metrics from past projects)
  - the degree to which the project plan reflects the abilities of the software team
  - the stability of product requirements and the environment that supports the software engineering effort.

# Functional Decomposition



# Conventional Methods: LOC/FP Approach

---

- compute LOC/FP using estimates of information domain values
- use historical data to build estimates for the project

# Example: LOC Approach

Function	Estimated LOC
user interface and control facilities (UICF)	2,300
two-dimensional geometric analysis (2D GA)	8,300
three-dimensional geometric analysis (3D GA)	6,800
database management (DBM)	3,300
computer graphics display facilities (CGDF)	4,900
peripheral control (PC)	2,100
design analysis modules (DAM)	8,400
<i>estimated lines of code</i>	<b>33,200</b>

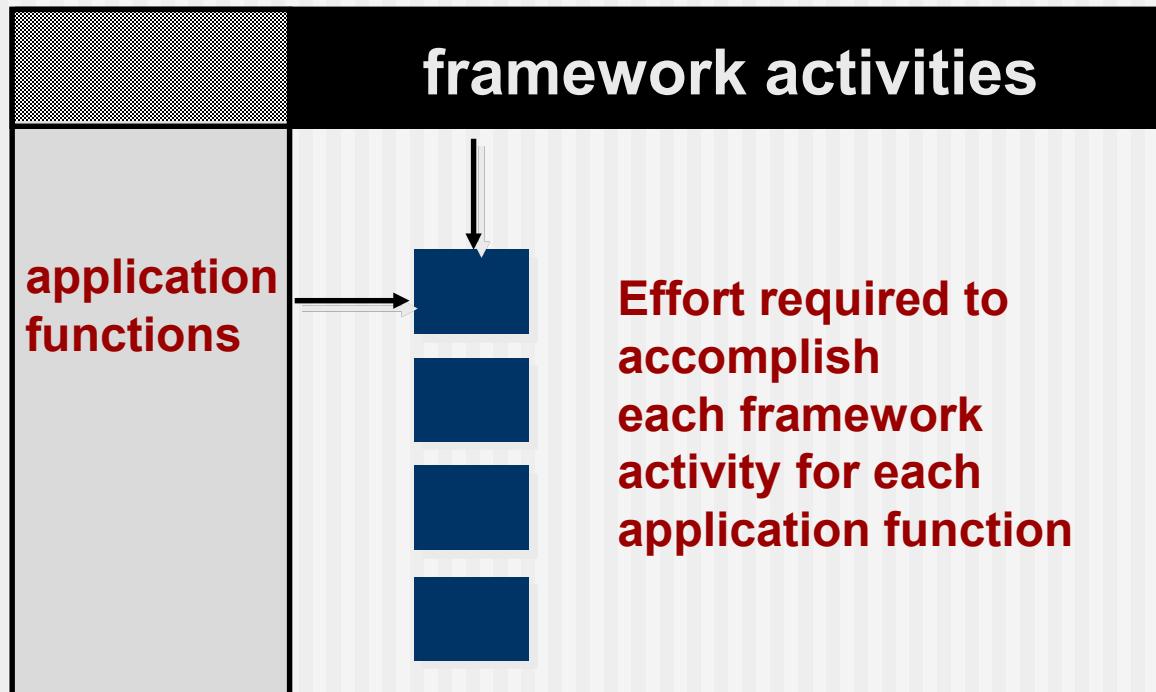
Average productivity for systems of this type = 620 LOC/pm.

Burdened labor rate = \$8000 per month, the cost per line of code is approximately \$13.

Based on the LOC estimate and the historical productivity data, the total estimated project cost is **\$431,000 and the estimated effort is 54 person-months.**

# Process-Based Estimation

Obtained from “process framework”



# Tool-Based Estimation

---

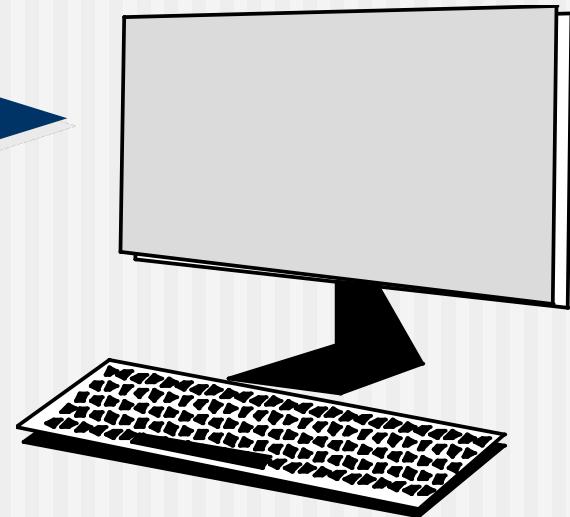
**project characteristics**



**calibration factors**



**LOC/FP data**



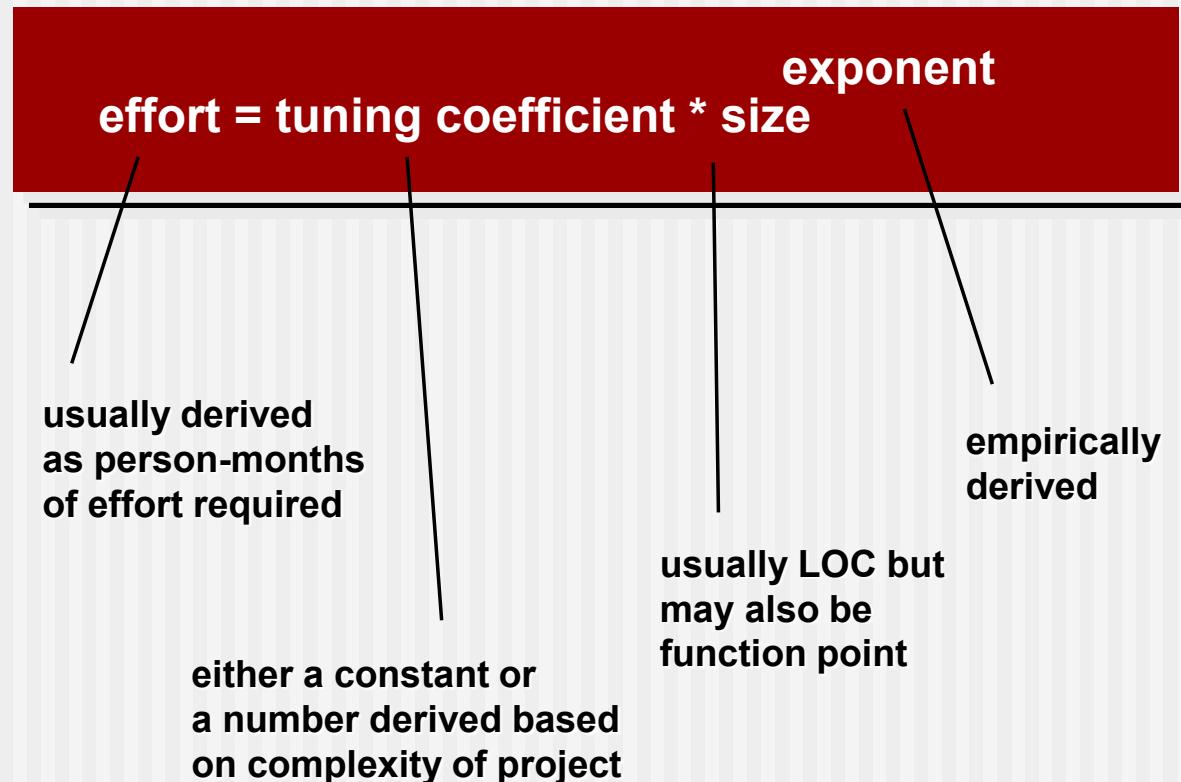
# Estimation with Use-Cases

	use cases	scenarios	pages	Estimated scenarios	pages	LOC	LOC estimate
User interface subsystem	6	10	6	12	5	560	3,366
Engineering subsystem group	10	20	8	16	8	3100	31,233
Infrastructure subsystem group	5	6	5	10	6	1650	7,970
Total LOC estimate				Estimated	Estimated	Estimated	42,568

Using 620 LOC/pm as the average productivity for systems of this type and a burdened labor rate of \$8000 per month, the cost per line of code is approximately \$13. Based on the use-case estimate and the historical productivity data, **the total estimated project cost is \$552,000 and the estimated effort is 68 person-months.**

# Empirical Estimation Models

*General form:*



# The Software Equation

---

*A dynamic multivariable model*

$$E = [LOC \times B^{0.333}/P]^3 \times (1/t^4)$$

where

E = effort in person-months or person-years

t = project duration in months or years

B = “special skills factor”

P = “productivity parameter”

# Estimation for OO Projects-I

---

- Develop estimates using effort decomposition, FP analysis, and any other method that is applicable for conventional applications.
- Using object-oriented requirements modeling (Chapter 6), develop use-cases and determine a count.
- From the analysis model, determine the number of key classes (called analysis classes in Chapter 6).
- Categorize the type of interface for the application and develop a multiplier for support classes:

<b>Interface type</b>	<b>Multiplier</b>
■ No GUI	2.0
■ Text-based user interface	2.25
■ GUI	2.5
■ Complex GUI	3.0

# Estimation for OO Projects-II

---

- Multiply the number of key classes (step 3) by the multiplier to obtain an estimate for the number of support classes.
- Multiply the total number of classes (key + support) by the average number of work-units per class. Lorenz and Kidd suggest 15 to 20 person-days per class.
- Cross check the class-based estimate by multiplying the average number of work-units per use-case

# Estimation for Agile Projects

---

- Each user scenario (a mini-use-case) is considered separately for estimation purposes.
- The scenario is decomposed into the set of software engineering tasks that will be required to develop it.
- Each task is estimated separately. Note: estimation can be based on historical data, an empirical model, or “experience.”
  - Alternatively, the ‘volume’ of the scenario can be estimated in LOC, FP or some other volume-oriented measure (e.g., use-case count).
- Estimates for each task are summed to create an estimate for the scenario.
  - Alternatively, the volume estimate for the scenario is translated into effort using historical data.
- The effort estimates for all scenarios that are to be implemented for a given software increment are summed to develop the effort estimate for the increment.

# Chapter 32

---

## ■ Process and Project Metrics

*Slide Set to accompany*

*Software Engineering: A Practitioner's Approach, 8/e*  
**by Roger S. Pressman and Bruce R. Maxim**

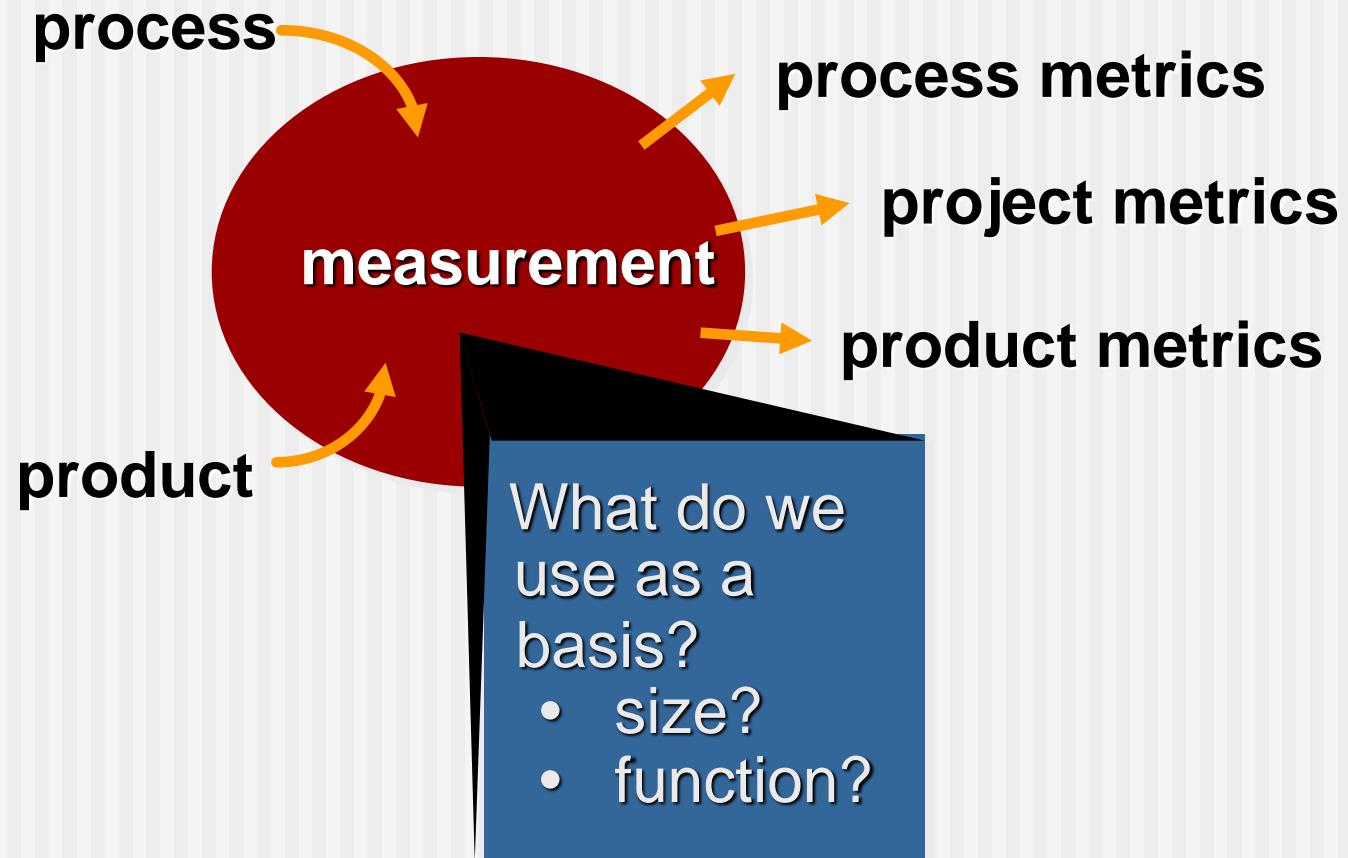
Slides copyright © 1996, 2001, 2005, 2009, 2014 by Roger S. Pressman

***For non-profit educational use only***

May be reproduced ONLY for student use at the university level when used in conjunction with *Software Engineering: A Practitioner's Approach, 8/e*. Any other reproduction or use is prohibited without the express written permission of the author.

All copyright information MUST appear if these slides are posted on a website for student use.

# A Good Manager Measures



# Why Do We Measure?

---

- assess the status of an ongoing project
- track potential risks
- uncover problem areas before they go “critical,”
- adjust work flow or tasks,
- evaluate the project team’s ability to control quality of software work products.

# Process Measurement

---

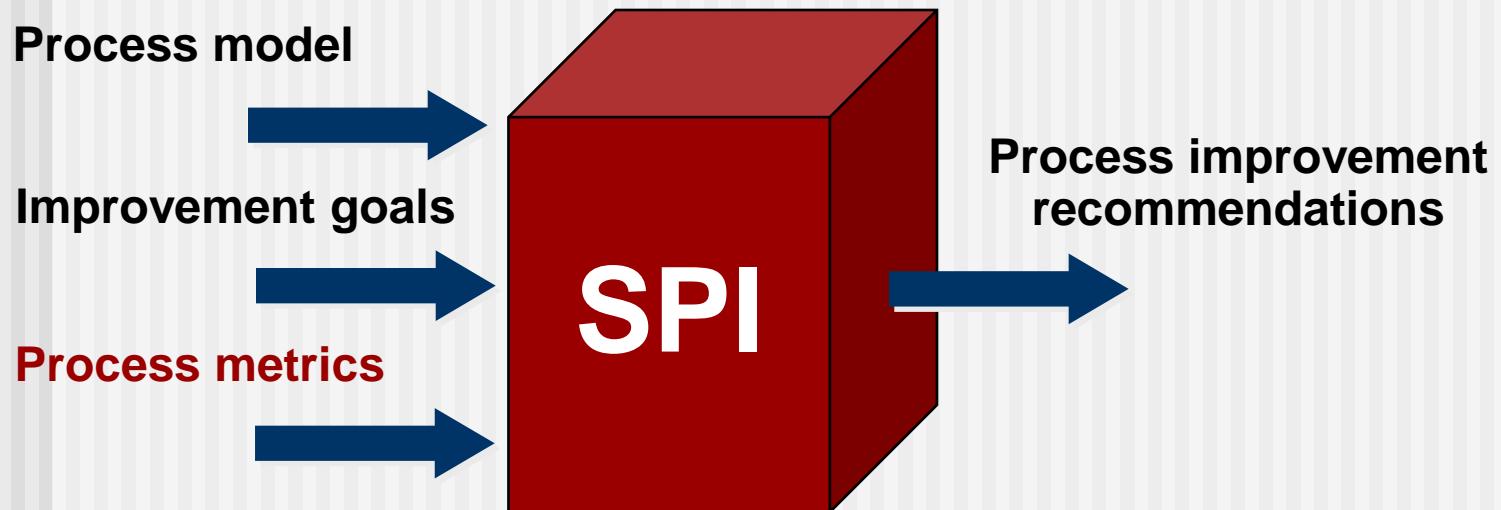
- We measure the efficacy of a software process indirectly.
  - That is, we derive a set of metrics based on the outcomes that can be derived from the process.
  - Outcomes include
    - measures of errors uncovered before release of the software
    - defects delivered to and reported by end-users
    - work products delivered (productivity)
    - human effort expended
    - calendar time expended
    - schedule conformance
    - other measures.
- We also derive process metrics by measuring the characteristics of specific software engineering tasks.

# Process Metrics Guidelines

---

- Use common sense and organizational sensitivity when interpreting metrics data.
- Provide regular feedback to the individuals and teams who collect measures and metrics.
- *Don't use metrics to appraise individuals.*
- Work with practitioners and teams to set clear goals and metrics that will be used to achieve them.
- *Never use metrics to threaten individuals or teams.*
- Metrics data that indicate a problem area should not be considered “negative.” These data are merely an indicator for process improvement.
- Don’t obsess on a single metric to the exclusion of other important metrics.

# Software Process Improvement



# Process Metrics

---

- **Quality-related**
  - focus on quality of work products and deliverables
- **Productivity-related**
  - Production of work-products related to effort expended
- **Statistical SQA data**
  - error categorization & analysis
- **Defect removal efficiency**
  - propagation of errors from process activity to activity
- **Reuse data**
  - The number of components produced and their degree of reusability

# Project Metrics

---

- used to minimize the development schedule by making the adjustments necessary to avoid delays and mitigate potential problems and risks
- used to assess product quality on an ongoing basis and, when necessary, modify the technical approach to improve quality.
- every project should measure:
  - *inputs*—measures of the resources (e.g., people, tools) required to do the work.
  - *outputs*—measures of the deliverables or work products created during the software engineering process.
  - *results*—measures that indicate the effectiveness of the deliverables.

# Typical Project Metrics

---

- Effort/time per software engineering task
- Errors uncovered per review hour
- Scheduled vs. actual milestone dates
- Changes (number) and their characteristics
- Distribution of effort on software engineering tasks

# Metrics Guidelines

---

- Use common sense and organizational sensitivity when interpreting metrics data.
- Provide regular feedback to the individuals and teams who have worked to collect measures and metrics.
- Don't use metrics to appraise individuals.
- Work with practitioners and teams to set clear goals and metrics that will be used to achieve them.
- Never use metrics to threaten individuals or teams.
- Metrics data that indicate a problem area should not be considered "negative." These data are merely an indicator for process improvement.
- Don't obsess on a single metric to the exclusion of other important metrics.

# Typical Size-Oriented Metrics

---

- errors per KLOC (thousand lines of code)
- defects per KLOC
- \$ per LOC
- pages of documentation per KLOC
- errors per person-month
- errors per review hour
- LOC per person-month
- \$ per page of documentation

# Typical Function-Oriented Metrics

---

- errors per FP (thousand lines of code)
- defects per FP
- \$ per FP
- pages of documentation per FP
- FP per person-month

# Comparing LOC and FP

Programming Language	LOC per Function point			
	avg.	median	low	high
Ada	154	-	104	205
Assembler	337	315	91	694
C	162	109	33	704
C++	66	53	29	178
COBOL	77	77	14	400
Java	63	53	77	-
JavaSc ript	58	63	42	75
Perl	60	-	-	-
PL/1	78	67	22	263
Powerbuilder	32	31	11	105
SAS	40	41	33	49
Smalltalk	26	19	10	55
SQL	40	37	7	110
Visual Basic	47	42	16	158

Representative values developed by QSM

# WebApp Project Metrics

---

- Number of **static Web pages** (the end-user has no control over the content displayed on the page)
- Number of **dynamic Web pages** (end-user actions result in customized content displayed on the page)
- Number of **internal page links** (internal page links are pointers that provide a hyperlink to some other Web page within the WebApp)
- Number of **persistent data objects**
- Number of **external systems interfaced**
- Number of **static content objects**
- Number of **dynamic content objects**
- Number of **executable functions**

# Measuring Quality

---

- **Correctness** — the degree to which a program operates according to specification
- **Maintainability**—the degree to which a program is amenable to change
- **Integrity**—the degree to which a program is impervious to outside attack
- **Usability**—the degree to which a program is easy to use

# Defect Removal Efficiency

---

$$\text{DRE} = E / (E + D)$$

*where:*

$E$  is the number of errors found before delivery of the software to the end-user

$D$  is the number of defects found after delivery.

# Metrics for Small Organizations

---

- time (hours or days) elapsed from the time a request is made until evaluation is complete,  $t_{queue}$ .
- effort (person-hours) to perform the evaluation,  $W_{eval}$ .
- time (hours or days) elapsed from completion of evaluation to assignment of change order to personnel,  $t_{eval}$ .
- effort (person-hours) required to make the change,  $W_{change}$ .
- time required (hours or days) to make the change,  $t_{change}$ .
- errors uncovered during work to make change,  $E_{change}$ .
- defects uncovered after change is released to the customer base,  $D_{change}$ .

# Chapter 35

---

## ■ Risk Analysis

*Slide Set to accompany*

*Software Engineering: A Practitioner's Approach, 8/e*  
**by Roger S. Pressman and Bruce R. Maxim**

Slides copyright © 1996, 2001, 2005, 2009, 2014 by Roger S. Pressman

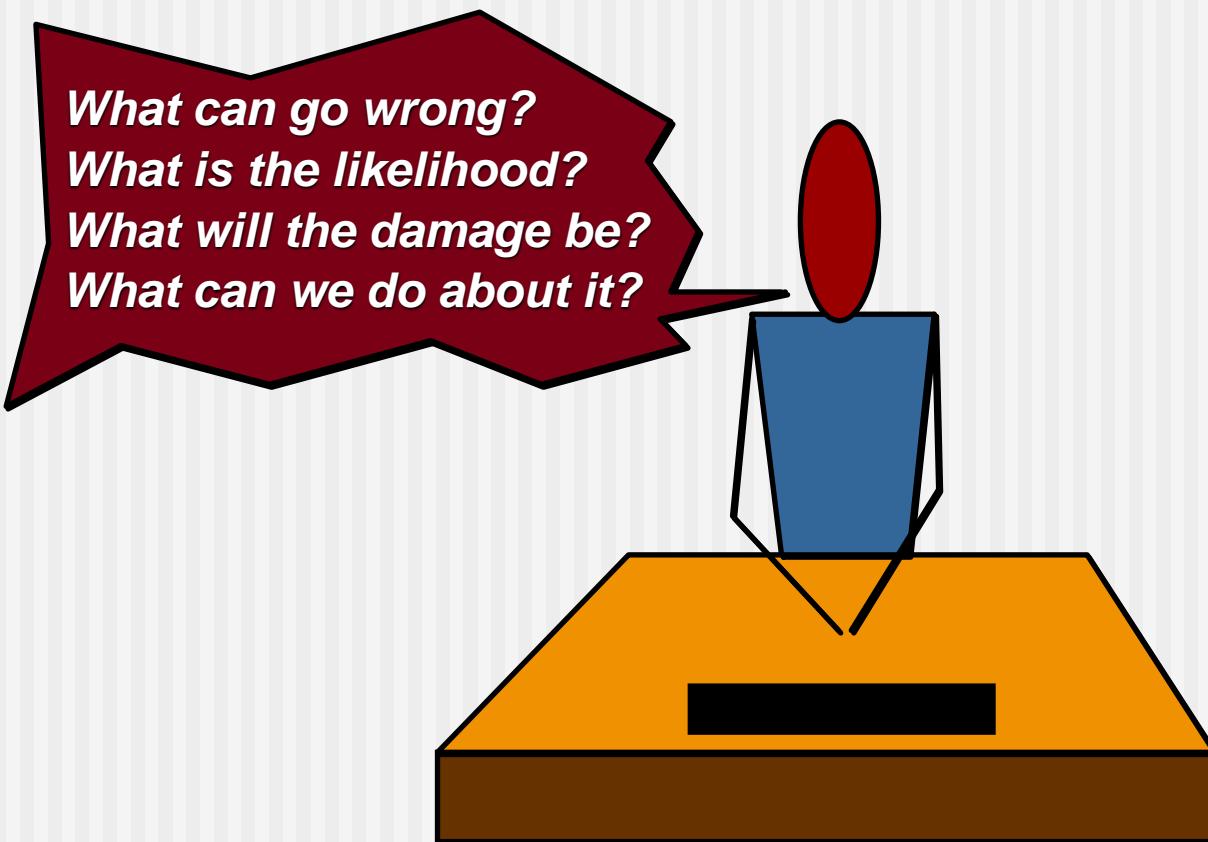
***For non-profit educational use only***

May be reproduced ONLY for student use at the university level when used in conjunction with *Software Engineering: A Practitioner's Approach, 8/e*. Any other reproduction or use is prohibited without the express written permission of the author.

All copyright information MUST appear if these slides are posted on a website for student use.

# Project Risks

---



# Reactive Risk Management

---

- project team reacts to risks when they occur
- mitigation—plan for additional resources in anticipation of fire fighting
- fix on failure—resource are found and applied when the risk strikes
- crisis management—failure does not respond to applied resources and project is in jeopardy

# Proactive Risk Management

---

- formal risk analysis is performed
- organization corrects the root causes of risk
  - TQM concepts and statistical SQA
  - examining risk sources that lie beyond the bounds of the software
  - developing the skill to manage change

# Seven Principles

---

- **Maintain a global perspective**—view software risks within the context of system and the business problem
- **Take a forward-looking view**—think about the risks that may arise in the future; establish contingency plans
- **Encourage open communication**—if someone states a potential risk, don't discount it.
- **Integrate**—a consideration of risk must be integrated into the software process
- **Emphasize a continuous process**—the team must be vigilant throughout the software process, modifying identified risks as more information is known and adding new ones as better insight is achieved.
- **Develop a shared product vision**—if all stakeholders share the same vision of the software, it likely that better risk identification and assessment will occur.
- **Encourage teamwork**—the talents, skills and knowledge of all stakeholder should be pooled

# Risk Management Paradigm



# Risk Identification

---

- *Product size*—risks associated with the overall size of the software to be built or modified.
- *Business impact*—risks associated with constraints imposed by management or the marketplace.
- *Customer characteristics*—risks associated with the sophistication of the customer and the developer's ability to communicate with the customer in a timely manner.
- *Process definition*—risks associated with the degree to which the software process has been defined and is followed by the development organization.
- *Development environment*—risks associated with the availability and quality of the tools to be used to build the product.
- *Technology to be built*—risks associated with the complexity of the system to be built and the "newness" of the technology that is packaged by the system.
- *Staff size and experience*—risks associated with the overall technical and project experience of the software engineers who will do the work.

# Assessing Project Risk-I

---

- Have top software and customer managers formally committed to support the project?
- Are end-users enthusiastically committed to the project and the system/product to be built?
- Are requirements fully understood by the software engineering team and their customers?
- Have customers been involved fully in the definition of requirements?
- Do end-users have realistic expectations?

# Assessing Project Risk-II

---

- Is project scope stable?
- Does the software engineering team have the right mix of skills?
- Are project requirements stable?
- Does the project team have experience with the technology to be implemented?
- Is the number of people on the project team adequate to do the job?
- Do all customer/user constituencies agree on the importance of the project and on the requirements for the system/product to be built?

# Risk Components

---

- ***performance risk***—the degree of uncertainty that the product will meet its requirements and be fit for its intended use.
- ***cost risk***—the degree of uncertainty that the project budget will be maintained.
- ***support risk***—the degree of uncertainty that the resultant software will be easy to correct, adapt, and enhance.
- ***schedule risk***—the degree of uncertainty that the project schedule will be maintained and that the product will be delivered on time.

# Risk Projection

---

- *Risk projection*, also called *risk estimation*, attempts to rate each risk in two ways
  - the likelihood or probability that the risk is real
  - the consequences of the problems associated with the risk, should it occur.
- There are four risk projection steps:
  - establish a scale that reflects the perceived likelihood of a risk
  - delineate the consequences of the risk
  - estimate the impact of the risk on the project and the product,
  - note the overall accuracy of the risk projection so that there will be no misunderstandings.

# Building a Risk Table

---

Risk	Probability	Impact	RMMM
			<b>Risk Mitigation Monitoring &amp; Management</b>

# Building the Risk Table

---

- Estimate the **probability** of occurrence
- Estimate the **impact** on the project on a scale of 1 to 5, where
  - 1 = low impact on project success
  - 5 = catastrophic impact on project success
- sort the table by probability and impact

# Risk Exposure (Impact)

---

The overall *risk exposure*, RE, is determined using the following relationship [Hal98]:

$$RE = P \times C$$

where

*P* is the probability of occurrence for a risk, and  
*C* is the cost to the project should the risk occur.

# Risk Exposure Example

---

- **Risk identification.** Only 70 percent of the software components scheduled for reuse will, in fact, be integrated into the application. The remaining functionality will have to be custom developed.
- **Risk probability.** 80% (likely).
- **Risk impact.** 60 reusable software components were planned. If only 70 percent can be used, 18 components would have to be developed from scratch (in addition to other custom software that has been scheduled for development). Since the average component is 100 LOC and local data indicate that the software engineering cost for each LOC is \$14.00, the overall cost (impact) to develop the components would be  $18 \times 100 \times 14 = \$25,200$ .
- **Risk exposure.**  $RE = 0.80 \times 25,200 \sim \$20,200$ .

# Risk Mitigation, Monitoring, and Management

---

- **mitigation**—how can we avoid the risk?
- **monitoring**—what factors can we track that will enable us to determine if the risk is becoming more or less likely?
- **management**—what contingency plans do we have if the risk becomes a reality?

# Risk Due to Product Size

---

## ***Attributes that affect risk:***

- estimated size of the product in LOC or FP?
- estimated size of product in number of programs, files, transactions?
- percentage deviation in size of product from average for previous products?
- size of database created or used by the product?
- number of users of the product?
- number of projected changes to the requirements for the product? before delivery? after delivery?
- amount of reused software?

# Risk Due to Business Impact

---

## ***Attributes that affect risk:***

- **affect of this product on company revenue?**
- **visibility of this product by senior management?**
- **reasonableness of delivery deadline?**
- **number of customers who will use this product**
- **interoperability constraints**
- **sophistication of end users?**
- **amount and quality of product documentation that must be produced and delivered to the customer?**
- **governmental constraints**
- **costs associated with late delivery?**
- **costs associated with a defective product?**

# Risks Due to the Customer

---

## ***Questions that must be answered:***

- Have you worked with the customer in the past?
- Does the customer have a solid idea of requirements?
- Has the customer agreed to spend time with you?
- Is the customer willing to participate in reviews?
- Is the customer technically sophisticated?
- Is the customer willing to let your people do their job—that is, will the customer resist looking over your shoulder during technically detailed work?
- Does the customer understand the software engineering process?

# Risks Due to Process Maturity

---

## ***Questions that must be answered:***

- Have you established a common process framework?
- Is it followed by project teams?
- Do you have management support for software engineering
- Do you have a proactive approach to SQA?
- Do you conduct formal technical reviews?
- Are CASE tools used for analysis, design and testing?
- Are the tools integrated with one another?
- Have document formats been established?

# Technology Risks

---

## ***Questions that must be answered:***

- Is the technology new to your organization?
- Are new algorithms, I/O technology required?
- Is new or unproven hardware involved?
- Does the application interface with new software?
- Is a specialized user interface required?
- Is the application radically different?
- Are you using new software engineering methods?
- Are you using unconventional software development methods, such as formal methods, AI-based approaches, artificial neural networks?
- Are there significant performance constraints?
- Is there doubt the functionality requested is "do-able?"

# Staff/People Risks

---

***Questions that must be answered:***

- Are the best people available?
- Does staff have the right skills?
- Are enough people available?
- Are staff committed for entire duration?
- Will some people work part time?
- Do staff have the right expectations?
- Have staff received necessary training?
- Will turnover among staff be low?

# Recording Risk Information

**Project:** Embedded software for XYZ system

**Risk type:** schedule risk

**Priority (1 low ... 5 critical):** 4

**Risk factor:** Project completion will depend on tests which require hardware component under development. Hardware component delivery may be delayed

**Probability:** 60 %

**Impact:** Project completion will be delayed for each day that hardware is unavailable for use in software testing

**Monitoring approach:**

Scheduled milestone reviews with hardware group

**Contingency plan:**

Modification of testing strategy to accommodate delay using software simulation

**Estimated resources:** 6 additional person months beginning in July

# Chapter 31

---

## ■ Project Management Concepts

*Slide Set to accompany*

*Software Engineering: A Practitioner's Approach, 8/e*  
by Roger S. Pressman and Bruce R. Maxim

Slides copyright © 1996, 2001, 2005, 2009, 2014 by Roger S. Pressman

***For non-profit educational use only***

May be reproduced ONLY for student use at the university level when used in conjunction with *Software Engineering: A Practitioner's Approach, 8/e*. Any other reproduction or use is prohibited without the express written permission of the author.

All copyright information MUST appear if these slides are posted on a website for student use.

# The Four P's

---

- **People** — the most important element of a successful project
- **Product** — the software to be built
- **Process** — the set of framework activities and software engineering tasks to get the job done
- **Project** — all work required to make the product a reality

# Stakeholders

---

- *Senior managers* who define the business issues that often have significant influence on the project.
- *Project (technical) managers* who must plan, motivate, organize, and control the practitioners who do software work.
- *Practitioners* who deliver the technical skills that are necessary to engineer a product or application.
- *Customers* who specify the requirements for the software to be engineered and other stakeholders who have a peripheral interest in the outcome.
- *End-users* who interact with the software once it is released for production use.

# Software Teams

---



# Team Leader

---

## ■ The MOI Model

- **Motivation.** The ability to encourage (by “push or pull”) technical people to produce to their best ability.
- **Organization.** The ability to mold existing processes (or invent new ones) that will enable the initial concept to be translated into a final product.
- **Ideas or innovation.** The ability to encourage people to create and feel creative even when they must work within bounds established for a particular software product or application.

# Software Teams

---

***The following factors must be considered when selecting a software project team structure ...***

- the difficulty of the problem to be solved
- the size of the resultant program(s) in lines of code or function points
- the time that the team will stay together (team lifetime)
- the degree to which the problem can be modularized
- the required quality and reliability of the system to be built
- the rigidity of the delivery date
- the degree of sociability (communication) required for the project

# Organizational Paradigms

---

- **closed paradigm**—structures a team along a traditional hierarchy of authority
- **random paradigm**—structures a team loosely and depends on individual initiative of the team members
- **open paradigm**—attempts to structure a team in a manner that achieves some of the controls associated with the closed paradigm but also much of the innovation that occurs when using the random paradigm
- **synchronous paradigm**—relies on the natural compartmentalization of a problem and organizes team members to work on pieces of the problem with little active communication among themselves

*suggested by Constantine [Con93]*

# Avoid Team “Toxicity”

---

- A frenzied work atmosphere in which team members waste energy and lose focus on the objectives of the work to be performed.
- High frustration caused by personal, business, or technological factors that cause friction among team members.
- “Fragmented or poorly coordinated procedures” or a poorly defined or improperly chosen process model that becomes a roadblock to accomplishment.
- Unclear definition of roles resulting in a lack of accountability and resultant finger-pointing.
- “Continuous and repeated exposure to failure” that leads to a loss of confidence and a lowering of morale.

# Agile Teams

---

- Team members must have trust in one another.
- The distribution of skills must be appropriate to the problem.
- Mavericks may have to be excluded from the team, if team cohesiveness is to be maintained.
- Team is “self-organizing”
  - An adaptive team structure
  - Uses elements of Constantine’s random, open, and synchronous paradigms
  - Significant autonomy

# Team Coordination & Communication

---

- *Formal, impersonal approaches* include software engineering documents and work products (including source code), technical memos, project milestones, schedules, and project control tools (Chapter 23), change requests and related documentation, error tracking reports, and repository data (see Chapter 26).
- *Formal, interpersonal procedures* focus on quality assurance activities (Chapter 25) applied to software engineering work products. These include status review meetings and design and code inspections.
- *Informal, interpersonal procedures* include group meetings for information dissemination and problem solving and “collocation of requirements and development staff.”
- *Electronic communication* encompasses electronic mail, electronic bulletin boards, and by extension, video-based conferencing systems.
- *Interpersonal networking* includes informal discussions with team members and those outside the project who may have experience or insight that can assist team members.

# The Product Scope

---

## ■ Scope

- **Context.** How does the software to be built fit into a larger system, product, or business context and what constraints are imposed as a result of the context?
- **Information objectives.** What customer-visible data objects (Chapter 8) are produced as output from the software? What data objects are required for input?
- **Function and performance.** What function does the software perform to transform input data into output? Are any special performance characteristics to be addressed?

## ■ Software project scope must be unambiguous and understandable at the management and technical levels.

# Problem Decomposition

---

- Sometimes called *partitioning* or *problem elaboration*
- Once scope is defined ...
  - It is decomposed into constituent functions
  - It is decomposed into user-visible data objects  
*or*
  - It is decomposed into a set of problem classes
- Decomposition process continues until all functions or problem classes have been defined

# The Process

---

- Once a process framework has been established
  - Consider project characteristics
  - Determine the degree of rigor required
  - Define a task set for each software engineering activity
    - Task set =
      - Software engineering tasks
      - Work products
      - Quality assurance points
      - Milestones

# Melding the Problem and the Process

These slides are designed to accompany *Software Engineering: A Practitioner's Approach*, 8/e (McGraw-Hill 2014). Slides copyright 2014 by Roger Pressman.

# The Project

---

- *Projects get into trouble when ...*
  - Software people don't understand their customer's needs.
  - The product scope is poorly defined.
  - Changes are managed poorly.
  - The chosen technology changes.
  - Business needs change [or are ill-defined].
  - Deadlines are unrealistic.
  - Users are resistant.
  - Sponsorship is lost [or was never properly obtained].
  - The project team lacks people with appropriate skills.
  - Managers [and practitioners] avoid best practices and lessons learned.

# Common-Sense Approach to Projects

---

- *Start on the right foot.* This is accomplished by working hard (very hard) to understand the problem that is to be solved and then setting realistic objectives and expectations.
- *Maintain momentum.* The project manager must provide incentives to keep turnover of personnel to an absolute minimum, the team should emphasize quality in every task it performs, and senior management should do everything possible to stay out of the team's way.
- *Track progress.* For a software project, progress is tracked as work products (e.g., models, source code, sets of test cases) are produced and approved (using formal technical reviews) as part of a quality assurance activity.
- *Make smart decisions.* In essence, the decisions of the project manager and the software team should be to "keep it simple."
- *Conduct a postmortem analysis.* Establish a consistent mechanism for extracting lessons learned for each project.

# To Get to the Essence of a Project

---

- Why is the system being developed?
- What will be done?
- When will it be accomplished?
- Who is responsible?
- Where are they organizationally located?
- How will the job be done technically and managerially?
- How much of each resource (e.g., people, software, tools, database) will be needed?

*Barry Boehm [Boe96]*

# Critical Practices

---

- Formal risk management
- Empirical cost and schedule estimation
- Metrics-based project management
- Earned value tracking
- Defect tracking against quality targets
- People aware project management