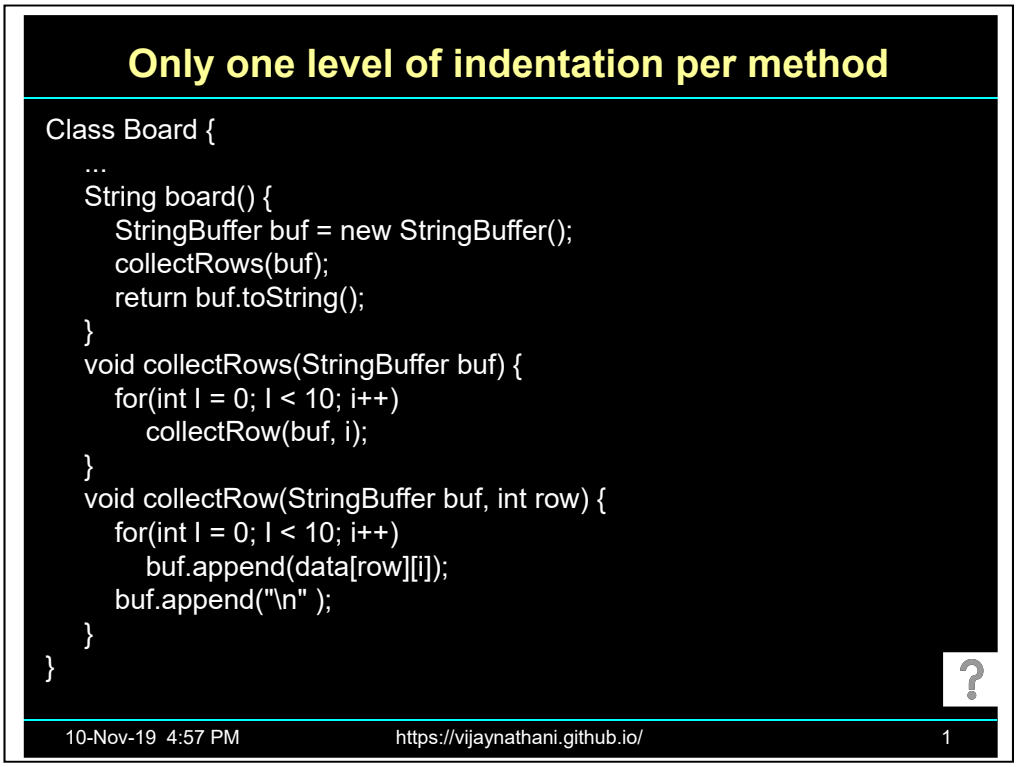


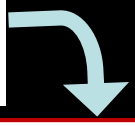
Self documenting code

Q20 – inch; Q06 – NO_GROUPING; Q07 – addHoliday; Q21 – full name in English;

Q22 – complexPassword; Q23 – TokenStream; Q25 - orderItems

Constants

```
for (int j=0; j<34; j++) {
    s += (t[j]*4)/5;
}
```



```
int realHoursPerIdealDay = 4;
const int WORK_DAYS_PER_WEEK = 5;
int sum = 0;
for (int j=0; j < NUMBER_OF_TASKS; j++) {
    int realTaskDays = taskEstimate[j] *
            realHoursPerIdealDay;
    int realTaskWeeks = realTaskDays /
            WORK_DAYS_PER_WEEK;
    sum += realTaskWeeks;
}
```

?

Q32 – FoodSalesReport.

Guidelines

- An interface should be designed so that it is easy to use and difficult to misuse.

API should be intuitive

- Size of String

```
myString.length(); //Java
myString.Length; //C#
length($my_string) #Perl
```

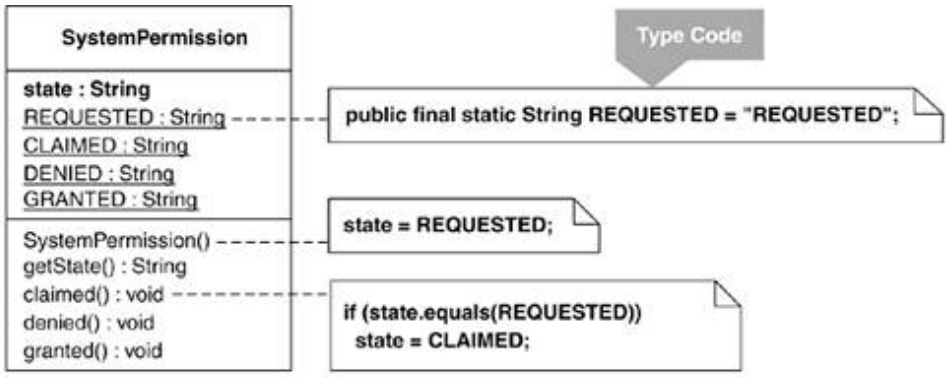
- Size of List

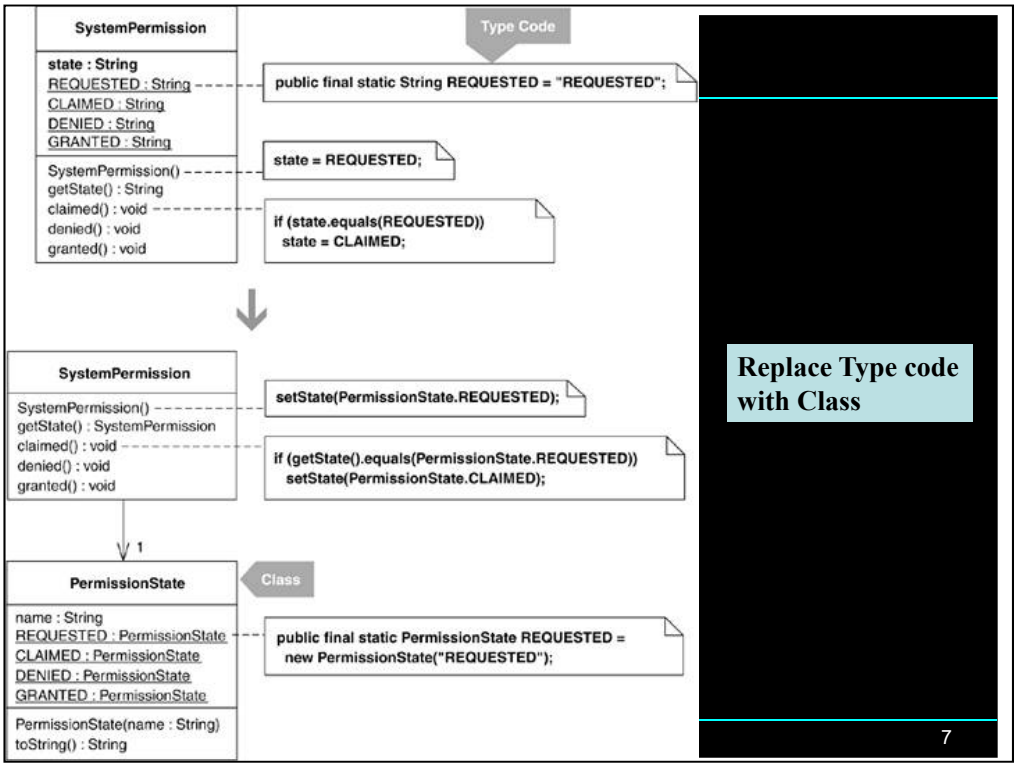
```
myList.size(); //Java
myList.Count; //C#
scalar(@my_list) #Perl
```

PHP String Library

- `str_repeat`
- `strcmp`
- `str_split`
- `strlen`
- `str_word_count`
- `strrev`

Improve Code





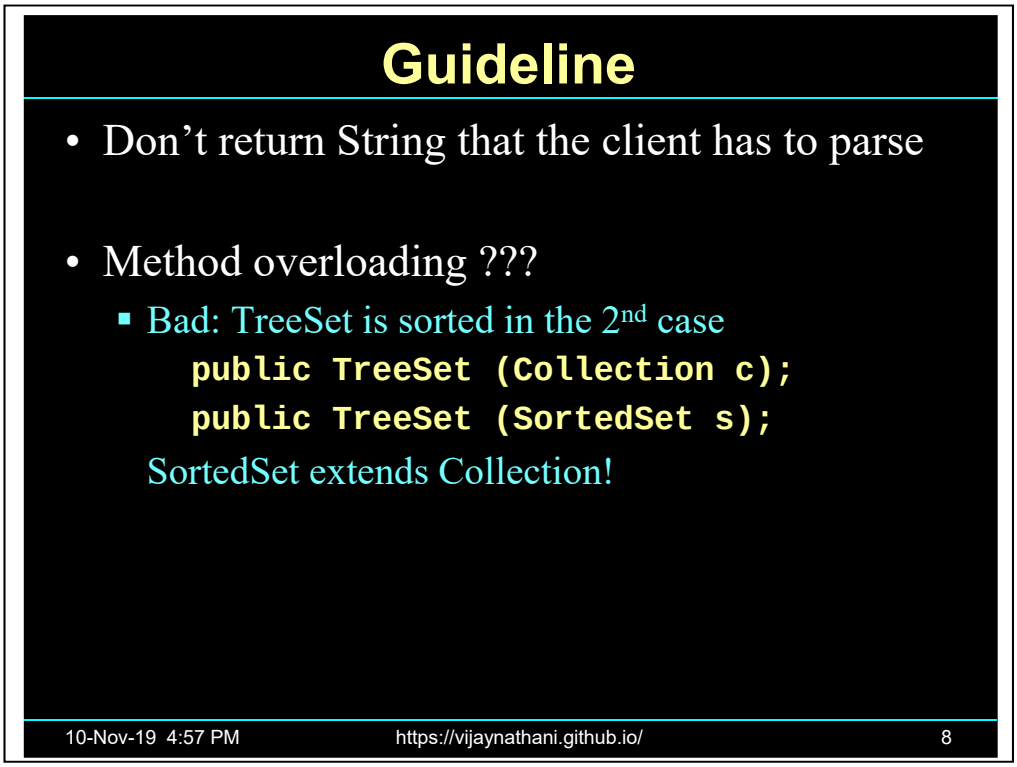
A field's type (e.g., a String or int) fails to protect it from unsafe assignments and invalid equality comparisons.

Constrain the assignments and equality comparisons by making the type of the field a class.

Benefits and Liabilities

\+ Provides better protection from invalid assignments and comparisons.

– Requires more code than using unsafe type does.



Return of sting with multiple values is bad because

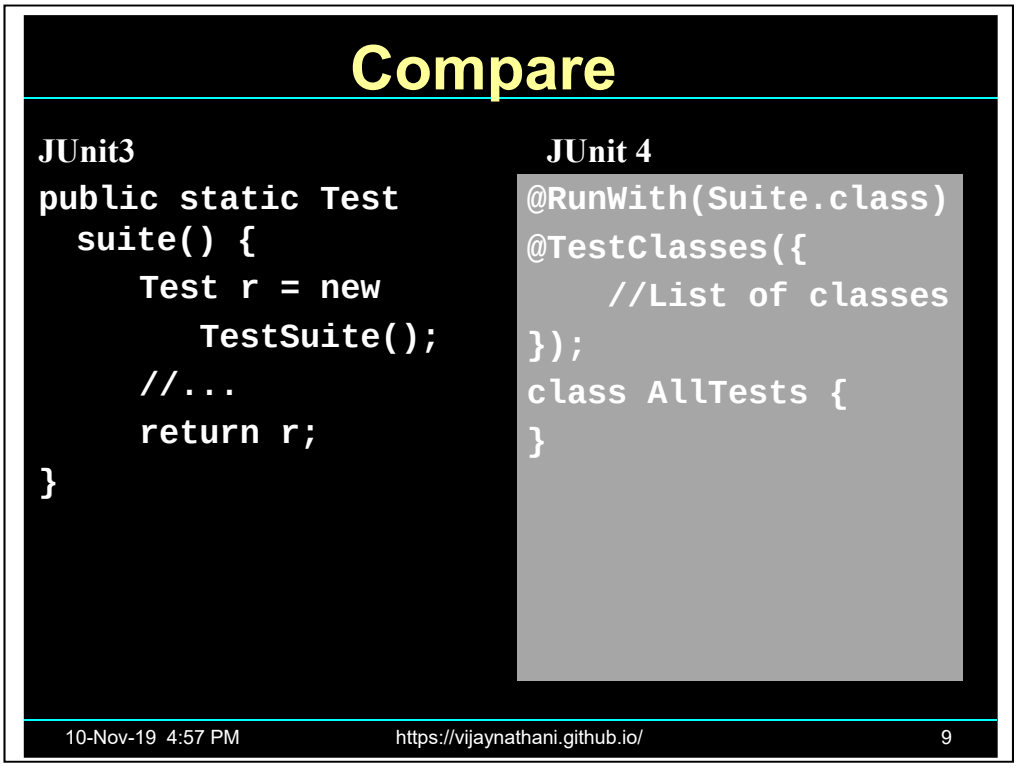
Future modifications will become difficult

Bad: In Java, printStackTrace of Throwable class

==========

Overloading - Avoid same name for multiple methods with same number of arguments

In C# and C++, operator overloading should be used only when it is clear. It should not be very frequent.



The code on the right

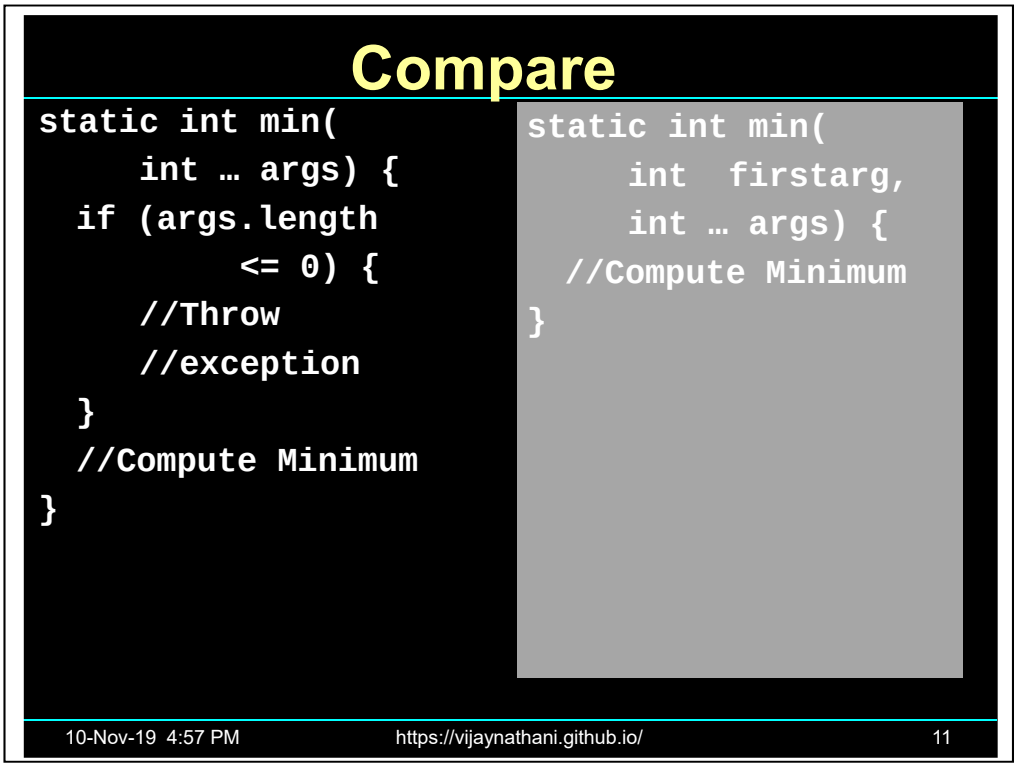
- Is flexible because any runner can be used
- Less prone to error, because the caller does not have to worry about exceptions

Improve

```
Collection<String> keys =
      new ArrayList<String>();
keys.add(key1);
keys.add(key2);
object.index(keys);
```

- Use varargs

```
object.index(key1, key2);
```

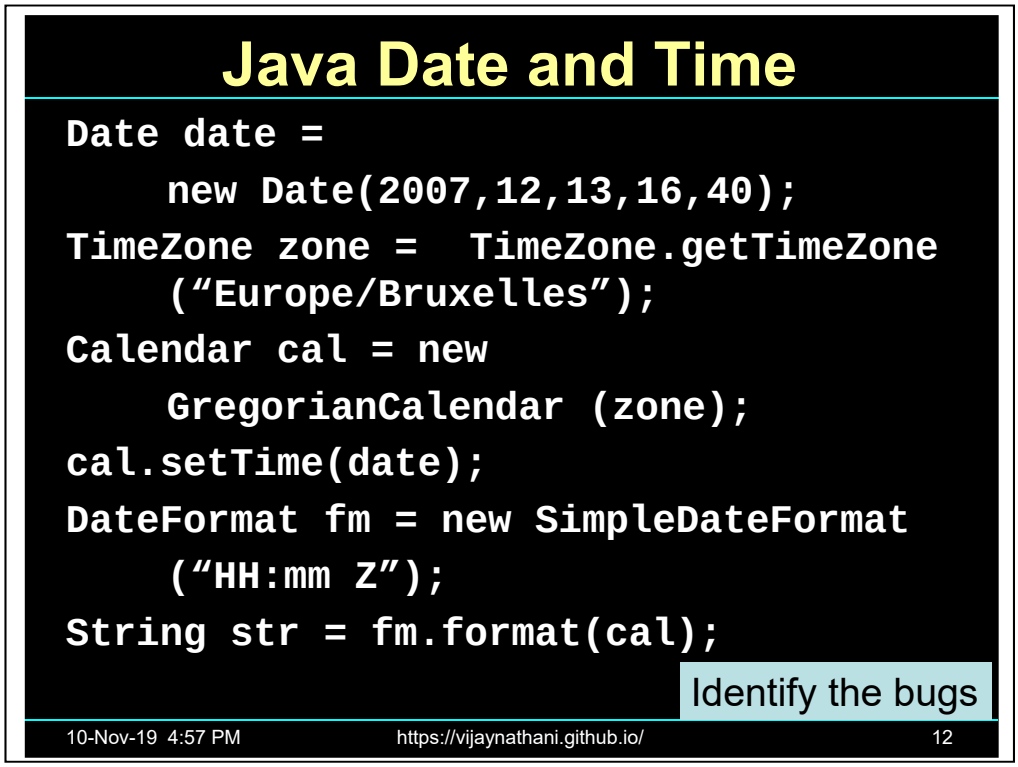


The code on the right

- Is flexible because any runner can be used
- Less prone to error, because the caller does not have to worry about exceptions

In C#, the syntax is

static int min(int firstarg, params int[] args)



Bug 1: Year should be 107, since base is 1900.

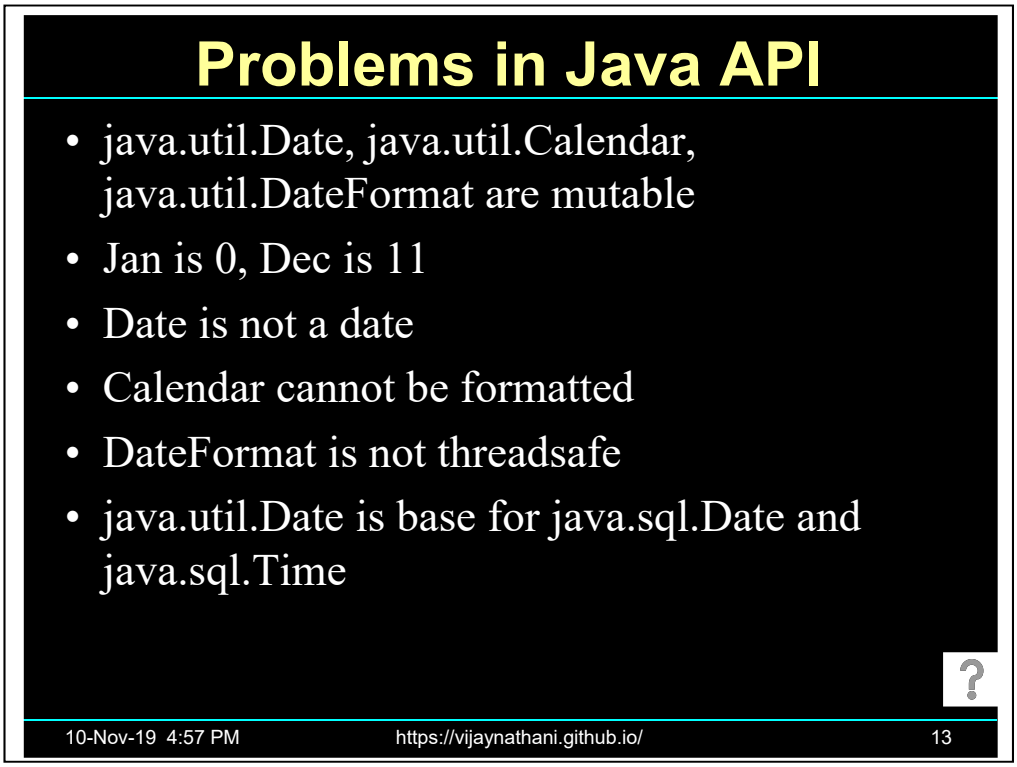
Bug 2: Month should be 11 instead of 12. Since Jan is 0.

Bug 3: “Europe/Bruxelles”. Bruzelles is capital of Brussels. It is capital of Belgium. Different people pronounce it different ways. Java returns GMT.

Bug 4: We are creating cal object with invalid or wrong value of date.

Not sure - Bug 5: fm.format gives runtime exception because it cannot format calendar. It can format only dates. So we need to call get “cal.getTime()” and pass the returned date to “fm.format”

Not sure - Bug 6: We have not set the timezone in DateFormat. It needs to be set before calling format.



Q53 – Piano

Q73 – Student, Teacher

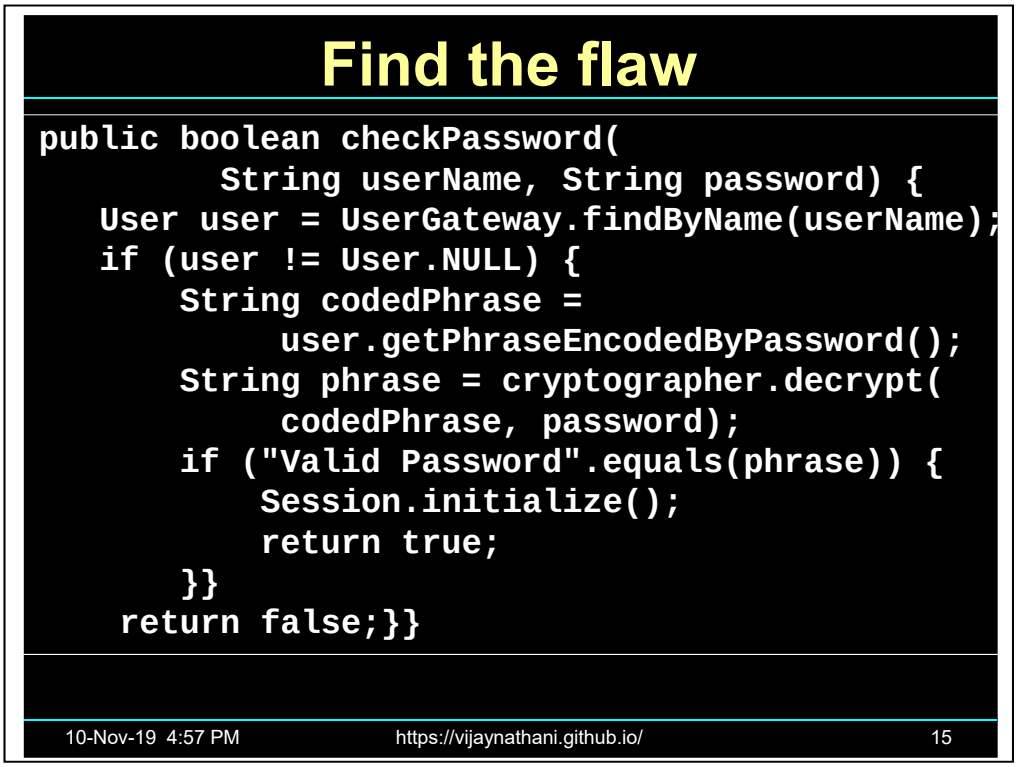
Date is not a date because: It has time & It uses from 1900

java.util.Date should not base class for java.sql.Date and Time because getYear on java.sql.Time throws an illegal argument exception.

Principle of Least Astonishment

- User of API should not be surprised by behavior
  - interrupted method in Thread class clears interrupted flag!

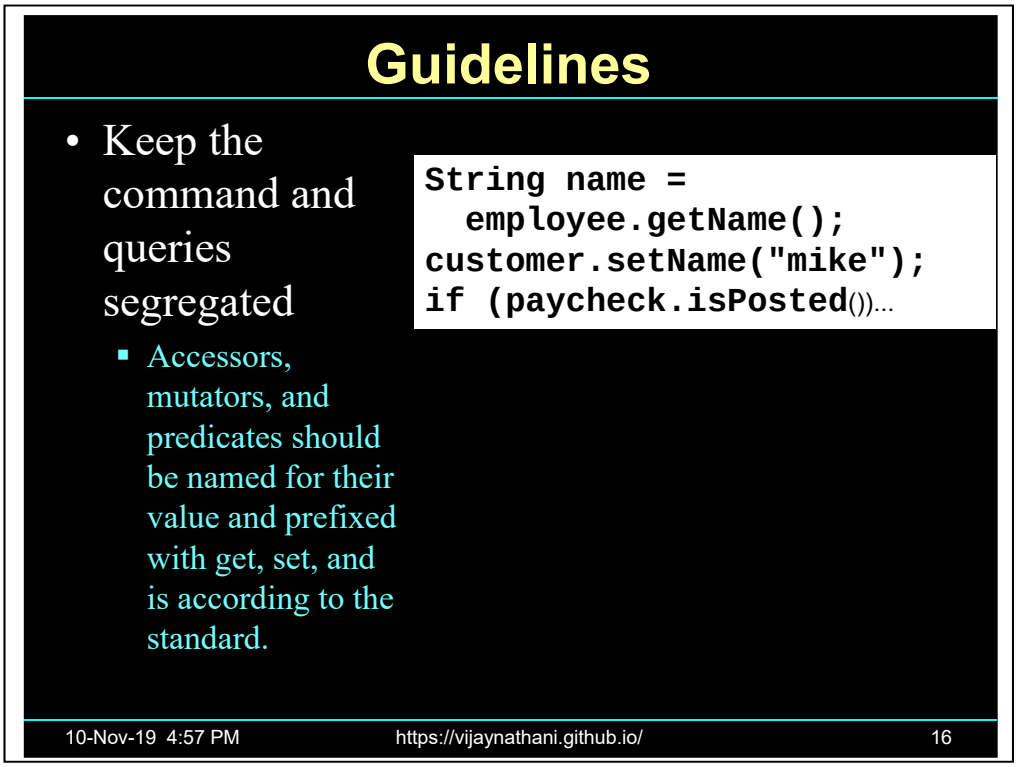
It should have been called clearInterruptedFlag



This function uses a standard algorithm to match a userName to a password. It returns true if they match and false if anything goes wrong. But it also has a side effect. Can you spot it?

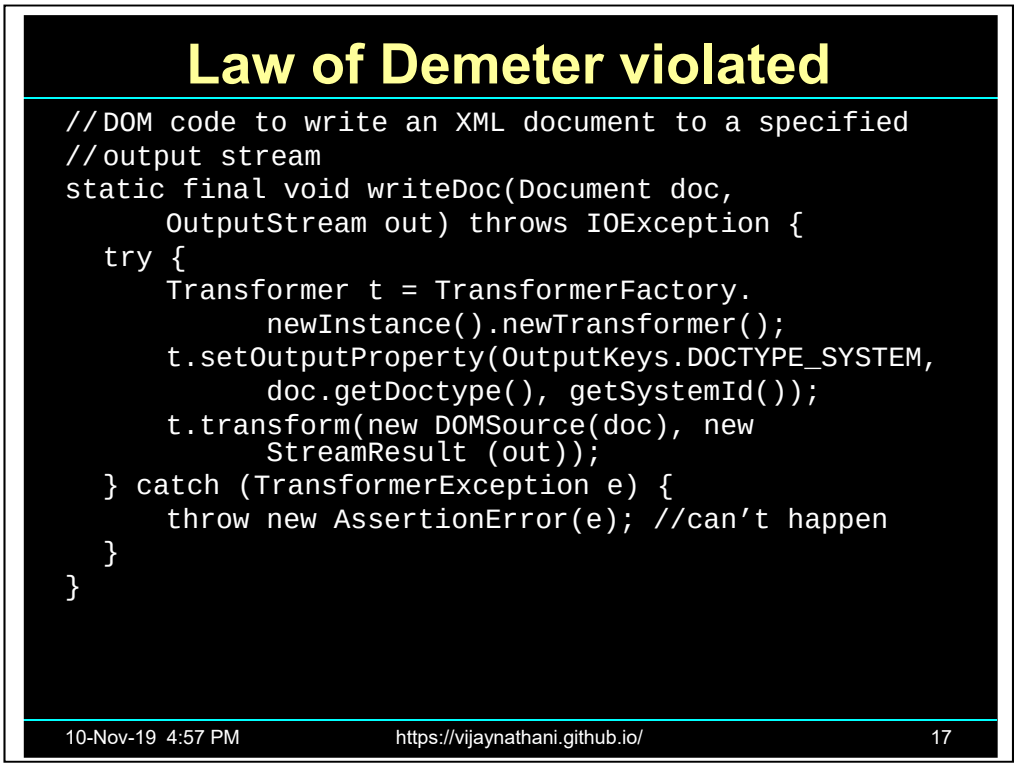
The side effect is the call to Session.initialize(), of course. The checkPassword function, by its name, says that it checks the password. The name does not imply that it initializes the session. So a caller who believes what the name of the function says runs the risk of erasing the existing session data when he or she decides to check the validity of the user.

This side effect creates a temporal coupling. That is, checkPassword can only be called at certain times (in other words, when it is safe to initialize the session). If it is called out of order, session data may be inadvertently lost. Temporal couplings are confusing, especially when hidden as a side effect. If you must have a temporal coupling, you should make it clear in the name of the function. In this case we might rename the function checkPasswordAndInitializeSession, though that certainly violates "Do one thing."



Exception: DSL.

Commands : return void.



Principle of least knowledge or Law of Demeter:

Each unit should only talk to its friends; Don't talk to strangers.

Don’t make the client do anything, that the Module could do

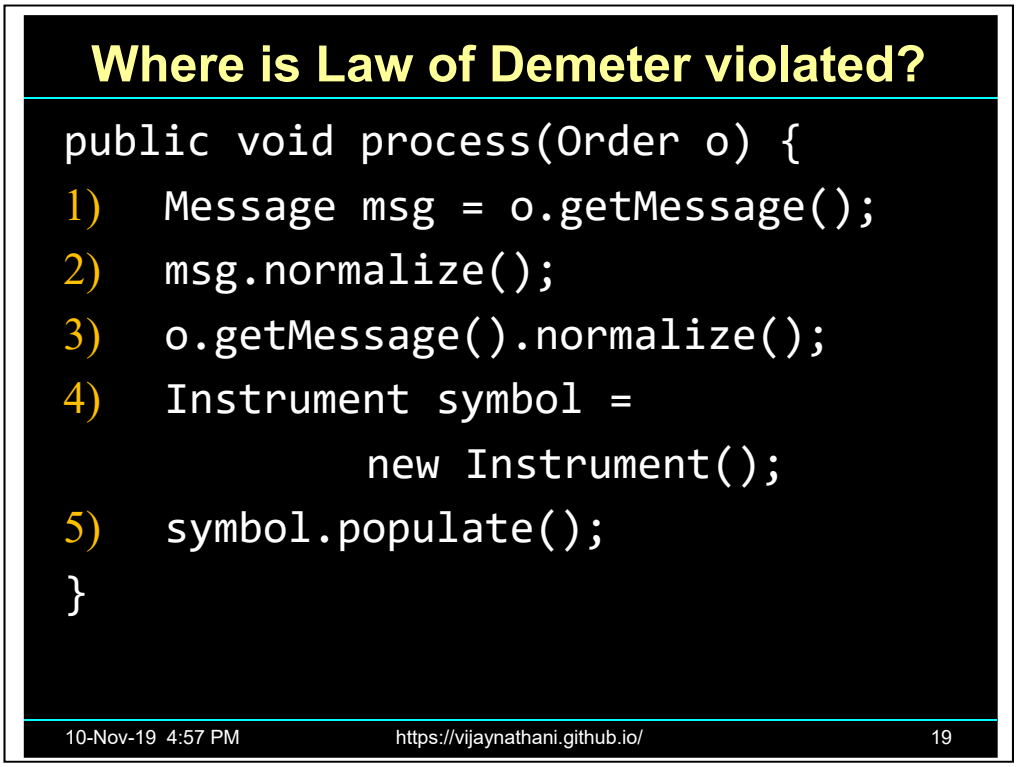
- Reduce the need for boilerplate code
- Generally done via cut-and-paste
- Ugly, Annoying and error-prone

Reduce Coupling

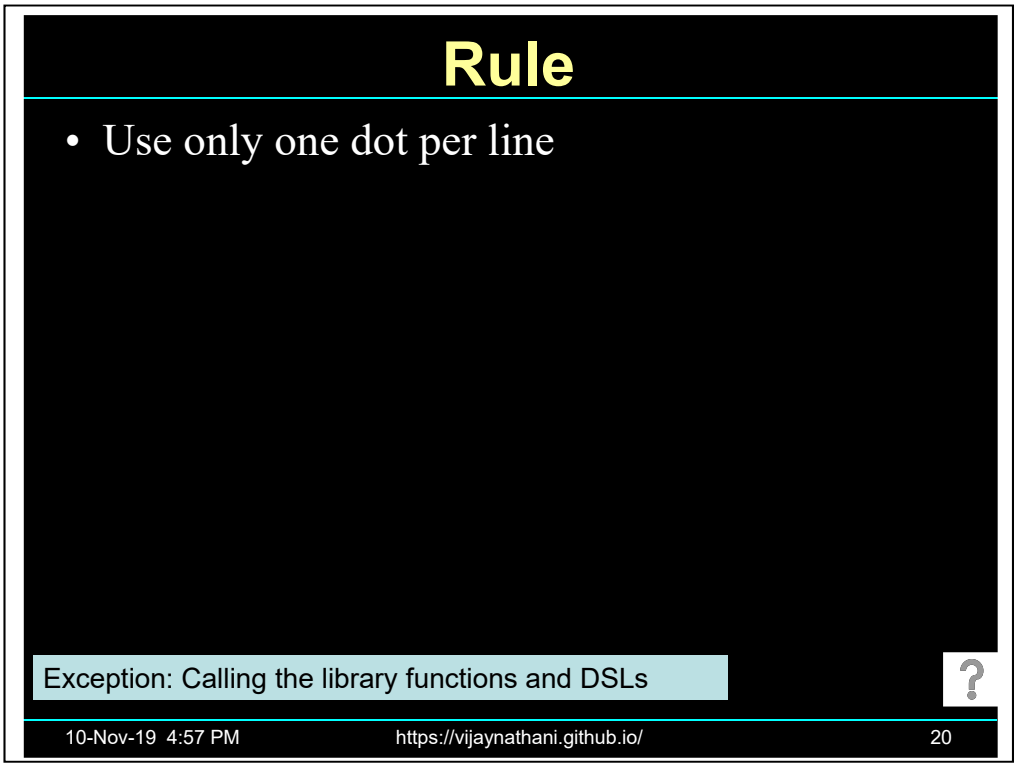
```
public float getTemp() {
  return
  station.getThermometer().getTemp();
}
```

Vs.

```
public float getTemp() {
  return station.getTemp();
}
```



Answer) lines 2 and 3


Rule
• Use only one dot per line
Exception: Calling the library functions and DSLs
10-Nov-19 4:57 PM
https://vijaynathani.github.io/
20

Q26Demeter.

Tell, Don’t Ask

- Ask for help, not information

10-Nov-19 4:57 PM https://vijaynathani.github.io/ 21

Never ask an object for information that you need to do something; rather, ask the object that has the information to do the work for you.

In other words: Don’t use any getters/setters/properties.

Avoid getters and setters

- Wrong

```
Money a, b, c;
//...
a.setValue( a.getValue() +
                b.getValue() );
```

- Right

```
Money a, b, c;
//...
a.increaseBy( b );
```

Improve

```
if (aCargo.getStatus() ==
          HandlingStatus.MISDIRECTED)
    ...

if (aCargo.isMisdirected())
    ...
```

Compare

```
Dog dog = new Dog();
dog.setBall(
      new Ball());
Ball ball =
      dog.getBAll());
```

```
Dog dog = new Dog();
dog.take(new Ball());
Ball ball =
      dog.give();
```

```
Dog dog = new Dog();
Dog.setWeight("23Kg");
```

```
Dog dog =
      new Dog("23Kg");
```

Example

Wrong:

```
MyThing[] things =
    thingManager.getThingList();
for (int i = 0; i < things.length; i++) {
    MyThing thing = things[i];
    if (thing.getName().equals(thingName))
        return thingManager.delete(thing);
}
```

Right:

```
    return thingManager.deleteThingNamed
        (thingName);
```

Fail-Fast

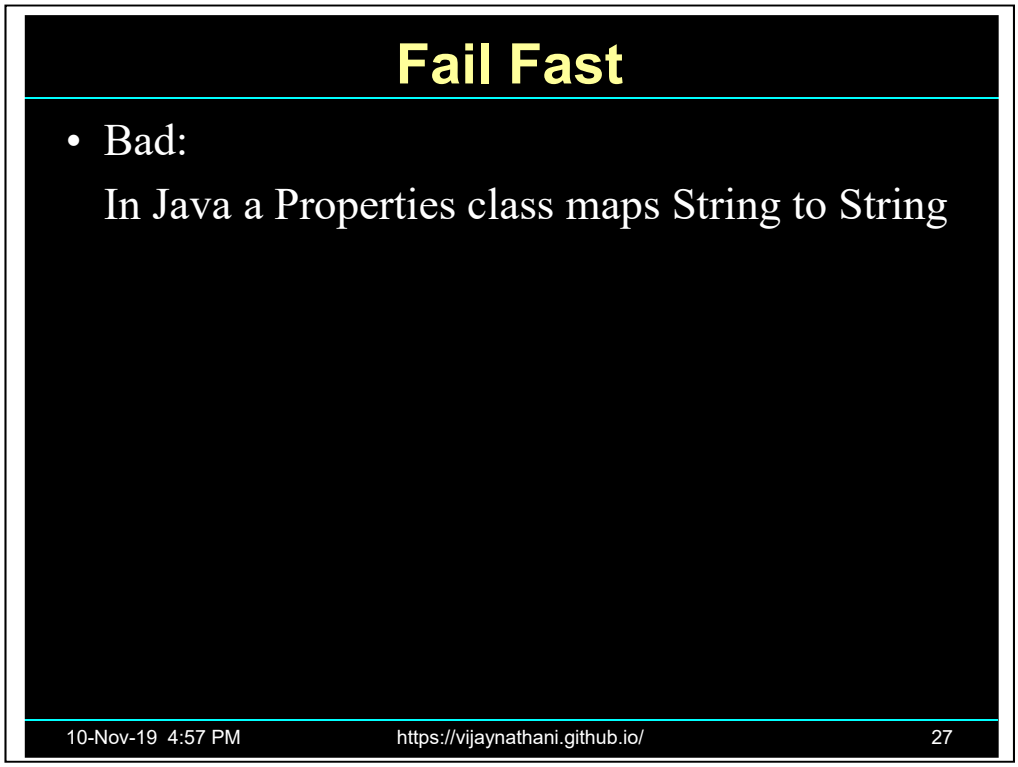
- Compile time checking is best

  Contrast this signature:

```
void assignCustomerToSalesman (
                long customerId,
                long salesmanId);
```

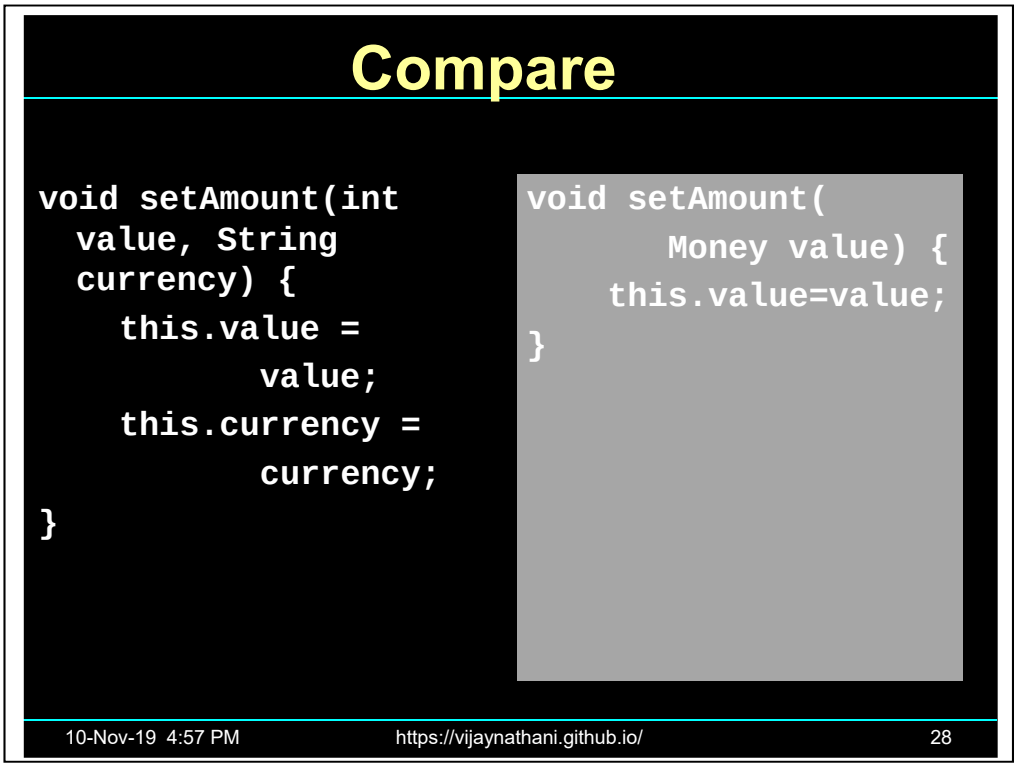
  with

```
void assignCustomerToSalesman (
                Customer c,
                Salesman s);
```



put method does not make this check

save method does this check



The code on the right

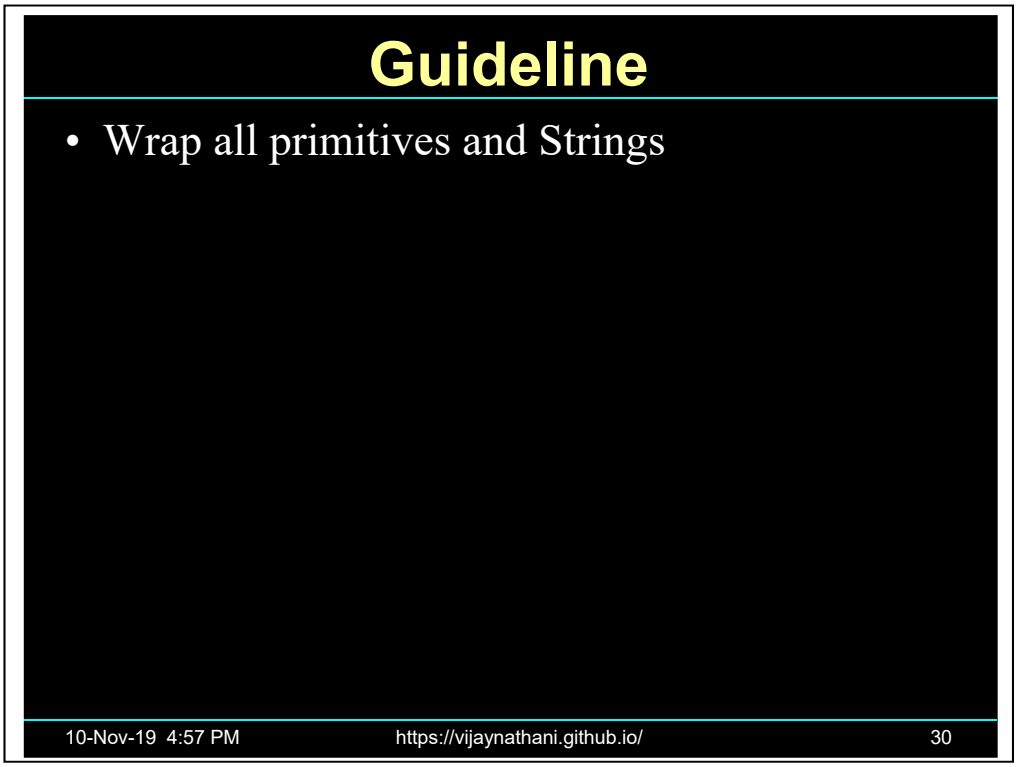
- Is flexible because any runner can be used
- Less prone to error, because the caller does not have to worry about exceptions

Improve

```
setOuterBounds ( x, y,
          width, height);
setInnerBounds ( x+2, y+2,
          width-4, height-4);
```

- Solution: Use Parameter Object

```
setOuterBounds (bounds);
setInnerBounds (bounds.expand
                  (-2));
```



An int on its own is just a scalar with no meaning. When a method takes an int as a parameter, the method name needs to do all the work of expressing the intent. If the same method takes an hour as a parameter, it's much easier to see what's happening. Small objects like this can make programs more maintainable, since it isn't possible to pass a year to a method that takes an hour parameter. With a primitive variable, the compiler can't help you write semantically correct programs. With an object, even a small one, you are giving both the compiler and the programmer additional information about what the value is and why it is being used.

Small objects such as hour or money also give you an obvious place to put behavior that otherwise would have been littered around other classes. This becomes especially true when you apply the rule relating to getters and setters and only the small object can access the value.

Direct access to Variables

- Compare

  `doorRegister=1;`

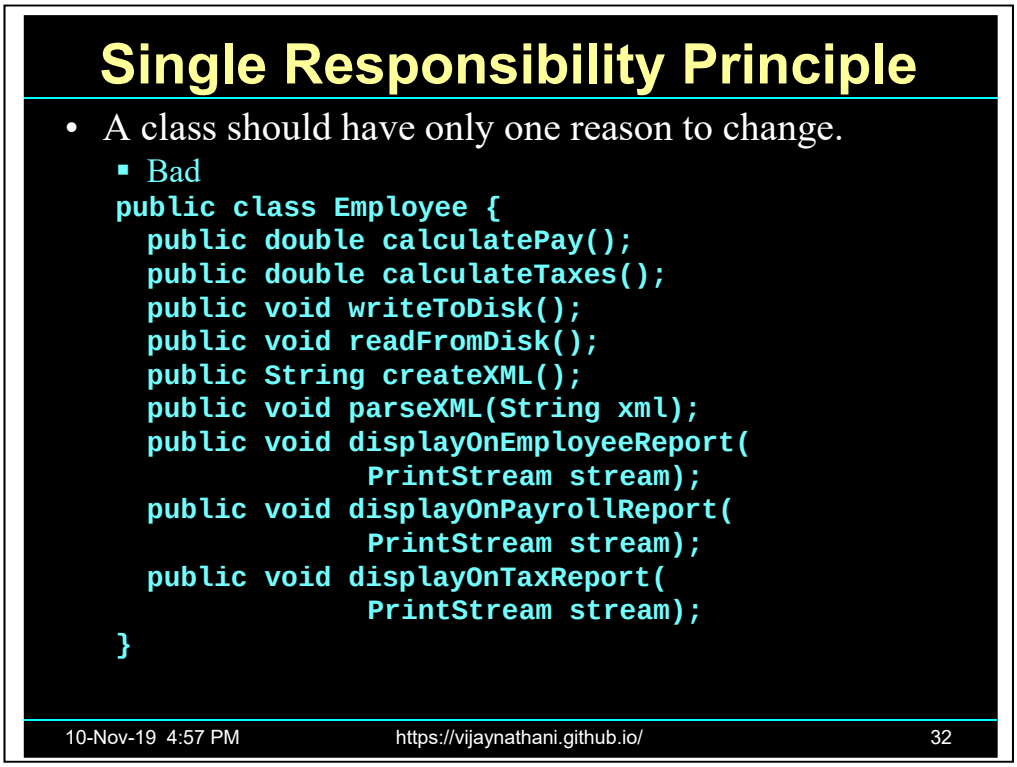
  with

  `openDoor();`

  with

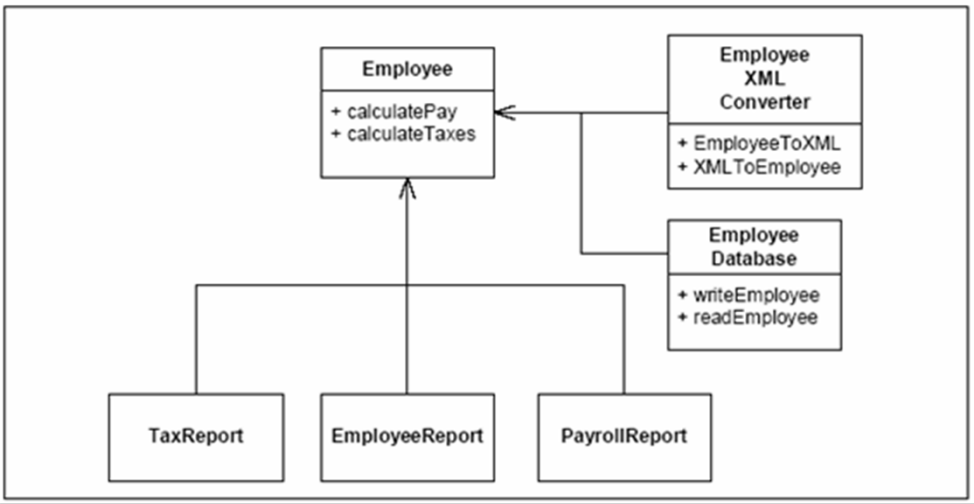
  `door.open();`

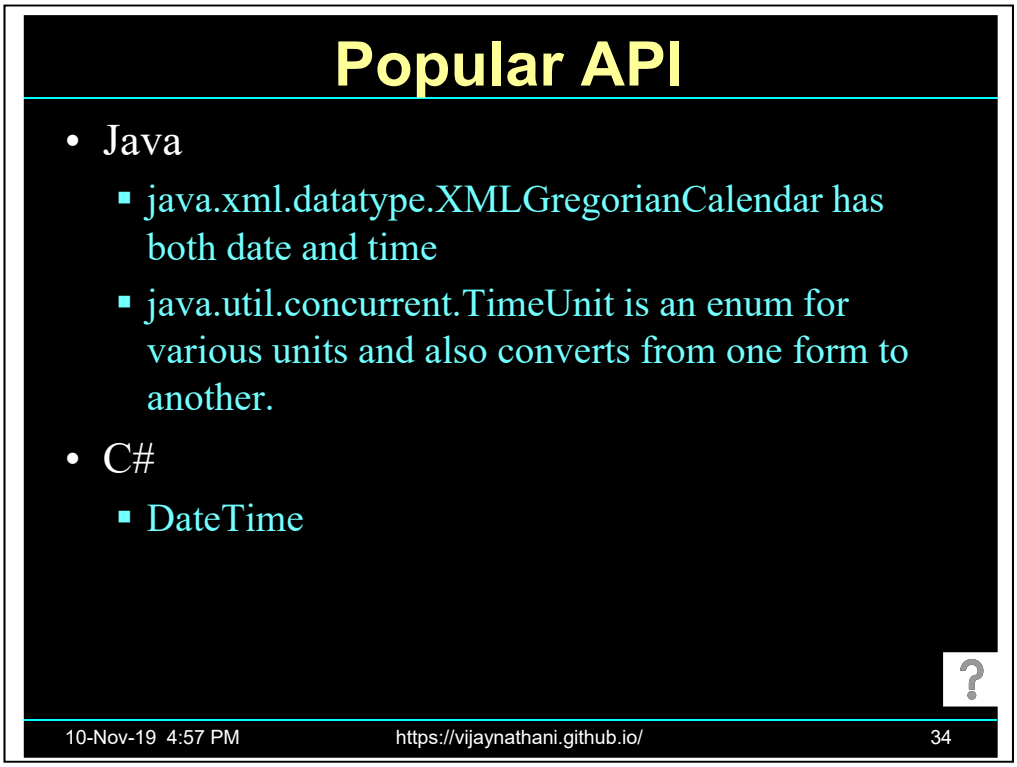
The last one is the best. It uses objects and functions.



An item such as a class should just have one responsibility and solve that responsibility well. If a class is responsible both for presentation and data access, that's a good example of a class breaking SRP.

SRP implemented

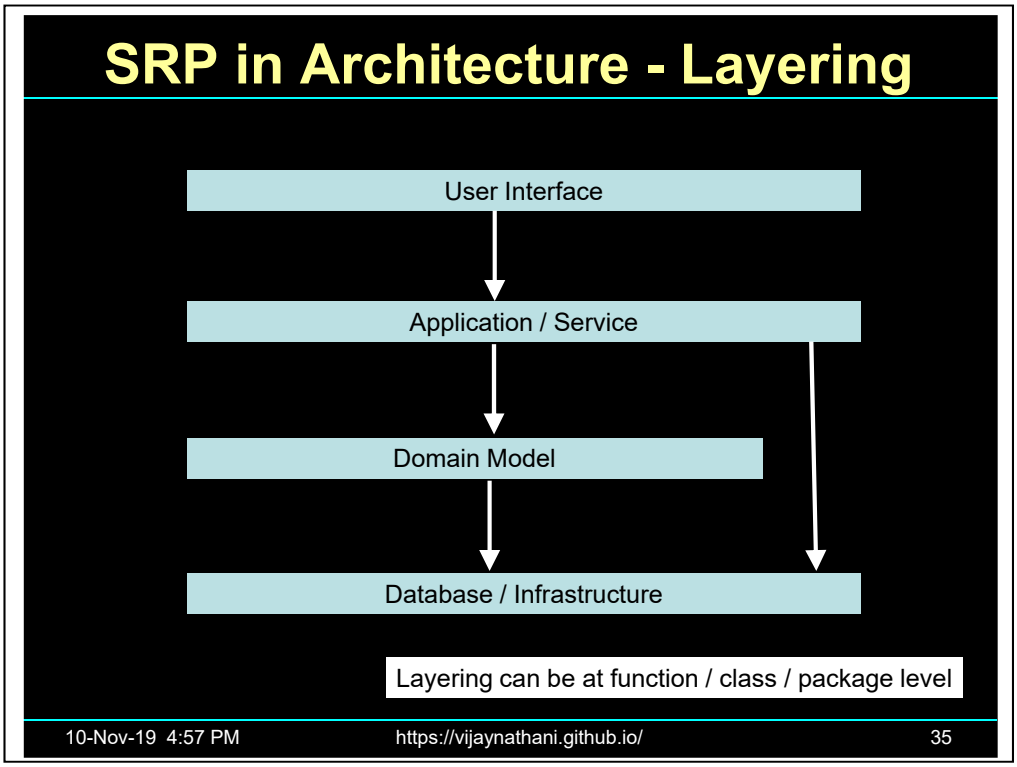




Q50 – Account

Q51 – Department

Q55 – processReport1



Service / Application layer co-ordinates tasks and delegates work to the domain. In an rich domain model, Service layer should be as thin as possible.

Interactions with the legacy systems is at Infrastructure level.

Usually the service layer

1. Gets the Database objects by interacting directly with Database layer and then
2. The domain model is executed in the objects.

In Java EE 1.4 or less,

Session beans are at service layer

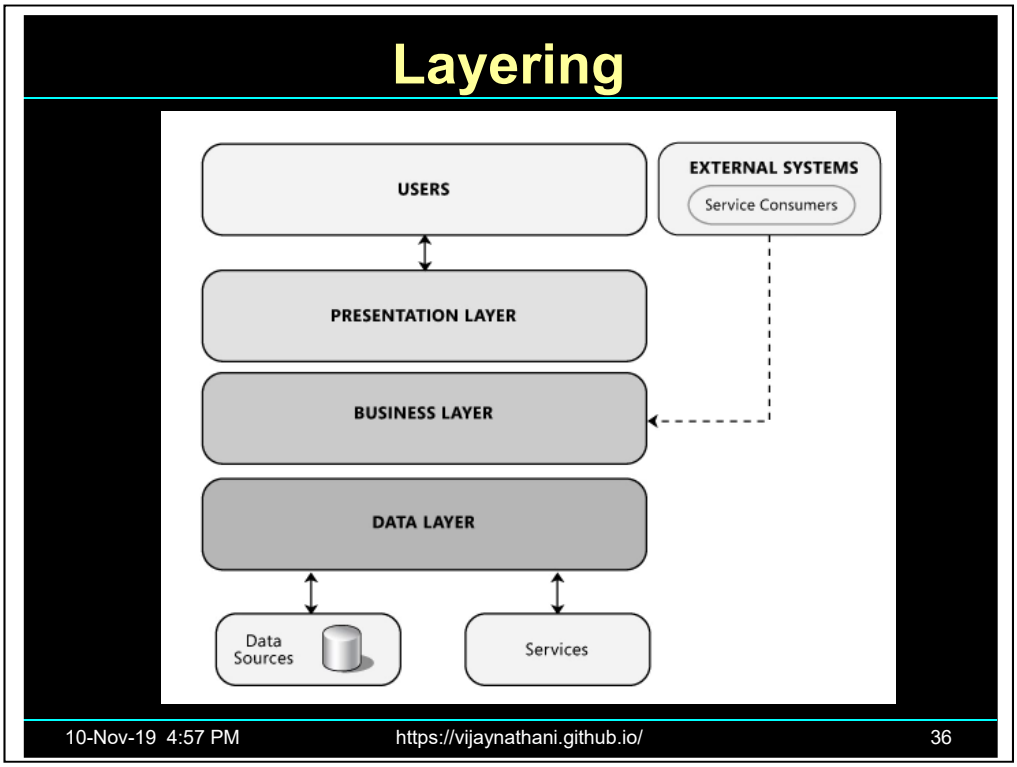
Entity beans are at Database layer.

If business logic code is kept in Session beans, then it will become transaction script.

If business logic code is kept in Entity beans, then it will become Table Module

Table module is popular in .NET. Transaction script was popular with VB6.

One of the hardest parts of working with domain logic seems to be that people often find it difficult to recognize what is domain logic and what is other forms of logic. An informal test I like is to imagine adding a radically different layer to an application, such as a command-line interface to a Web application. If there's any functionality you have to duplicate in order to do this, that's a sign of where domain logic has leaked into the presentation. Similarly, do you have to duplicate logic to replace a relational database with an XML file?



Layers are logical separation. Tiers are physical separation. It is possible to run all the four layers in the same tier.

A rich domain model needs Transaction management, Security enforcement and Remote management.

==========

Low coupling between layers. High cohesion between them.

User interface modules should not contain business logic.

No circular references between layers.

Business layer uses abstraction of technological services.

Layers should be testable individually.

Layers are a logical abstract and not necessarily physical.

Layers should be shy about their internals.

Interface Segregation Principle

- Interfaces should be as fine-grained as possible.
  - Any problem:

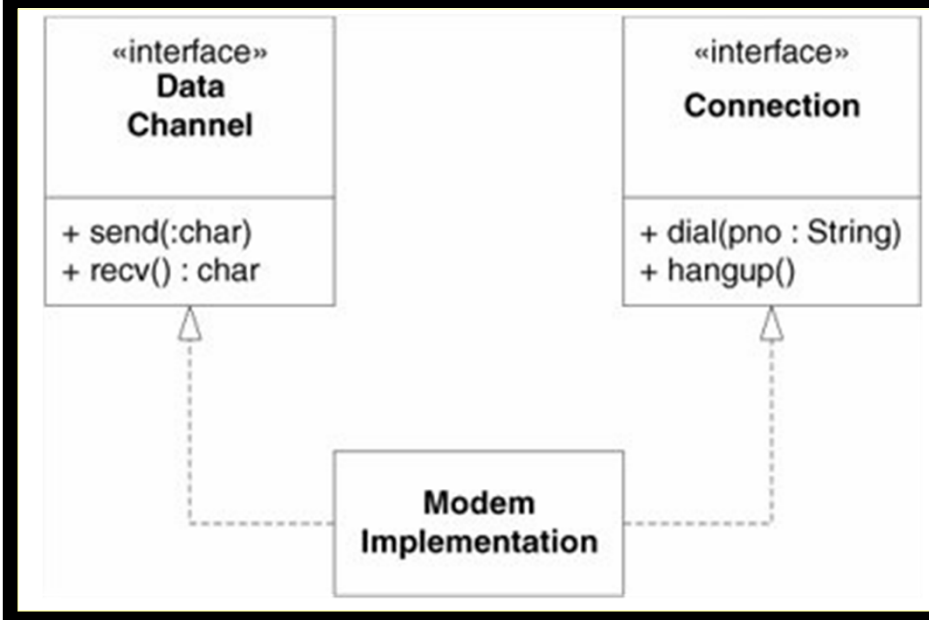
```
public interface Modem {
  public void dial(String pno);
  public void hangup();
  public void send(Char c);
  public char recv();
}
```

10-Nov-19 4:57 PM https://vijaynathani.github.io/ 37

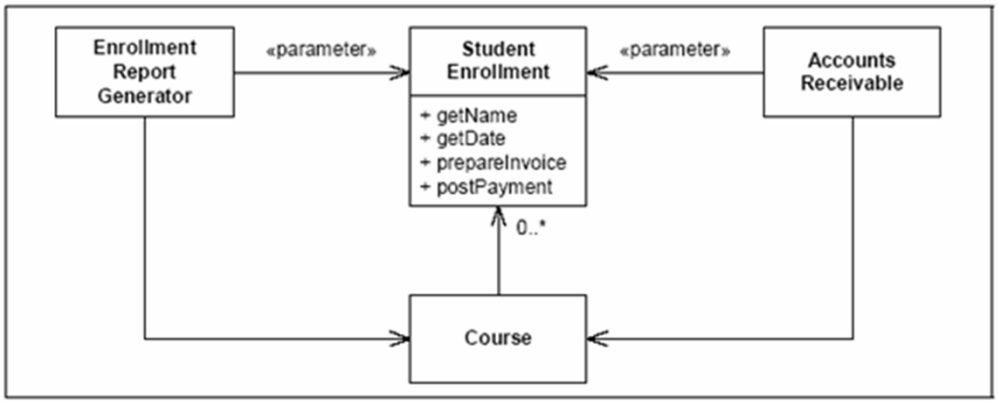
Clients should not be forced to depend upon the interfaces that they do no use. – Robert Martin

If a class implements an interface with multiple methods, but in one of the methods throws notSupportedException, then this principle is violated.

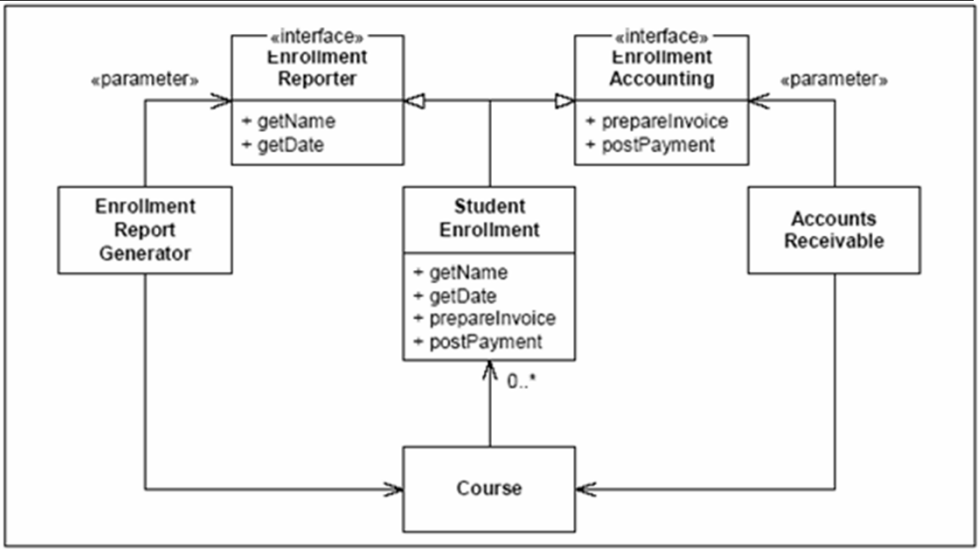
ISP implemented



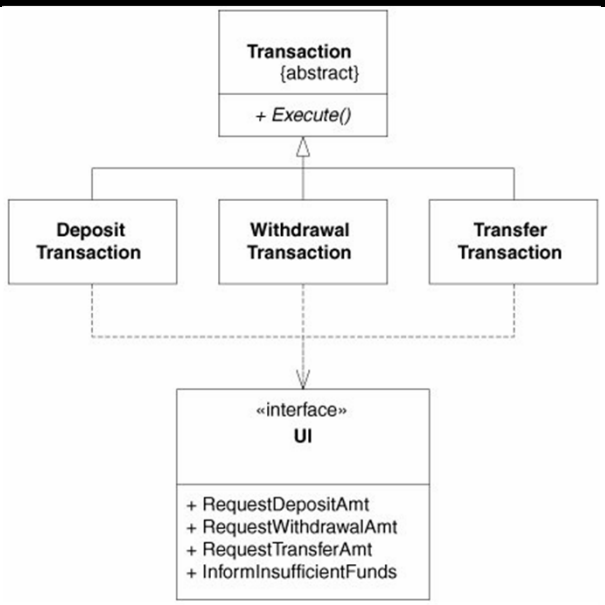
ISP violated



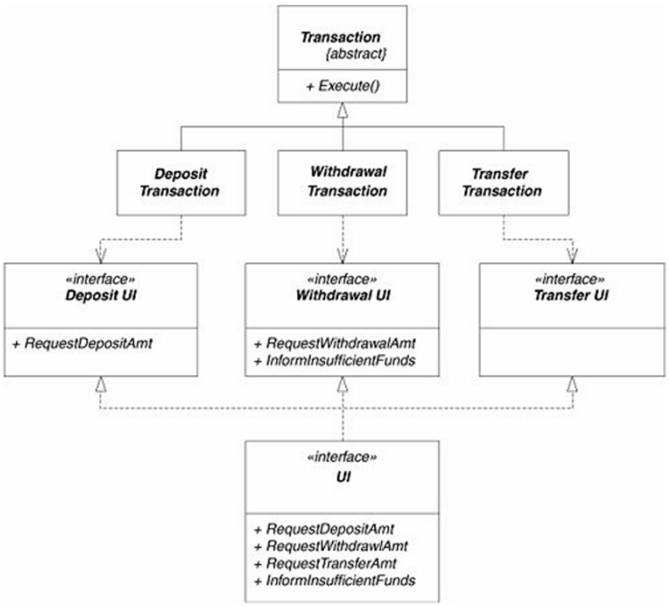
ISP implemented

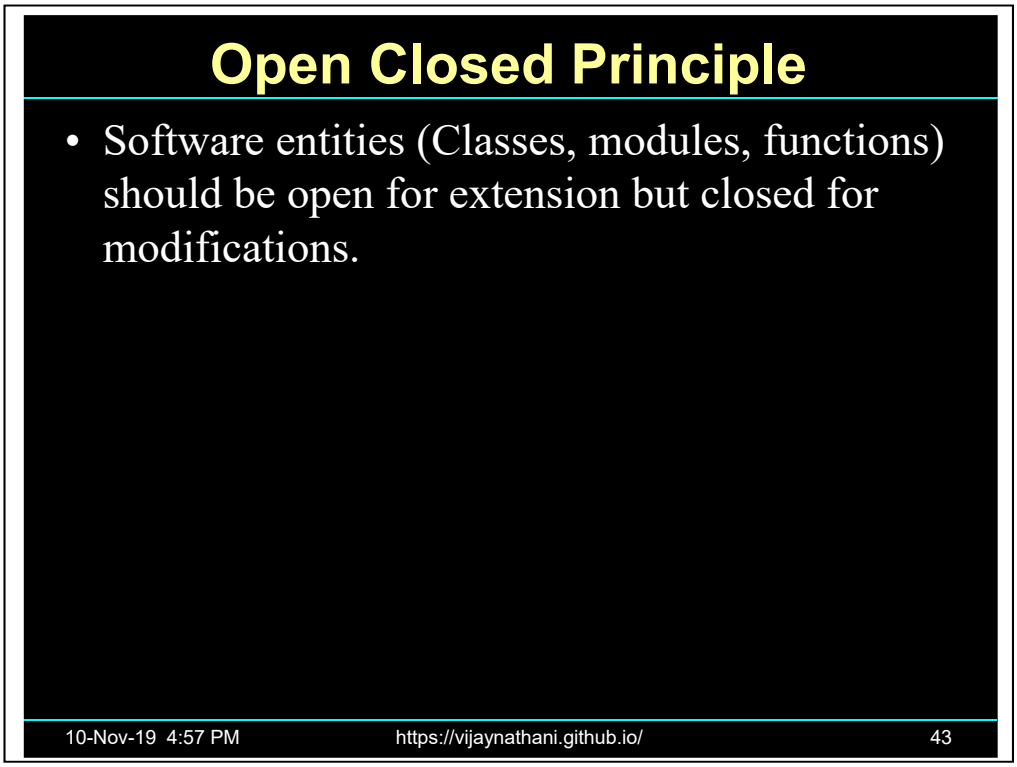


ISP violated



ISP implemented





It should be possible to change the environment of a class without changing the class.

A class should be closed for modification, but open for extension. When you change a class, there is always a risk that you will break something. But if instead of modifying the class you extend it with a sub-class, that's a less risky change.

Guidelines

- Keep things that vary separately from things that are common.

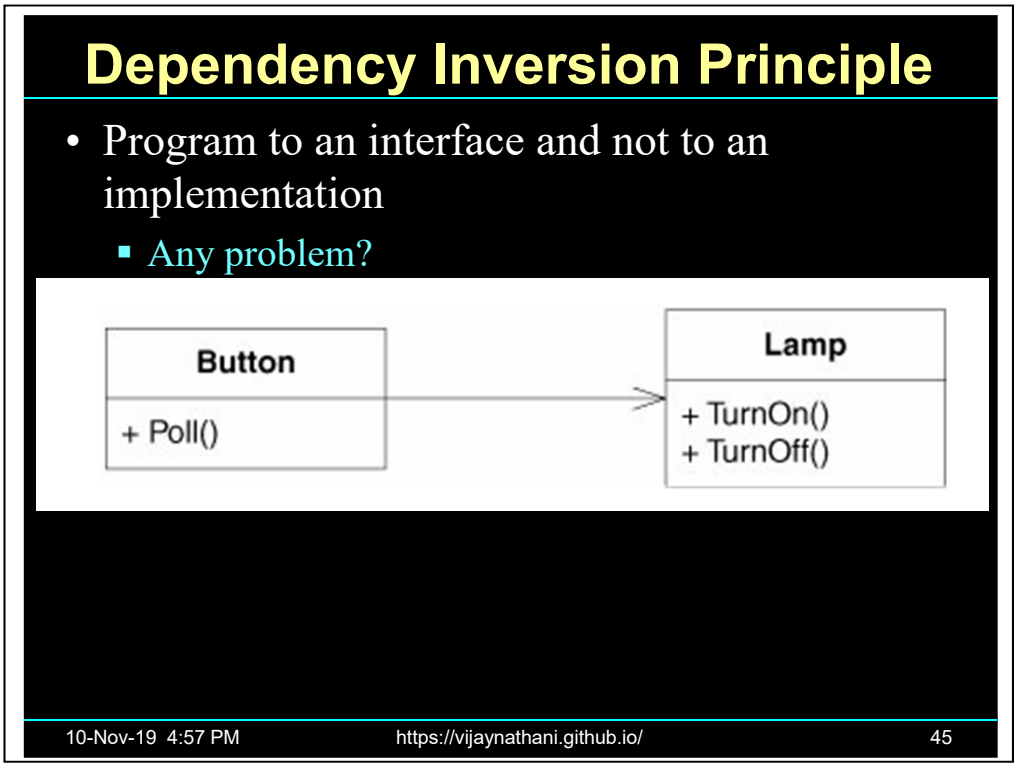
Q45 – LoanHandler

Q44 – FILE1, DATABASE1

Q39 - Scheduler

Q83 – Cooker

Q84 – ChooseFontDialog



"HIGH LEVEL MODULES SHOULD NOT DEPEND UPON LOW LEVEL MODULES, BOTH SHOULD DEPEND UPON ABSTRACTIONS.

ABSTRACTIONS SHOULD NOT DEPEND UPON DETAILS. DETAILS SHOULD DEPEND UPON ABSTRACTIONS."

No variable should hold a reference to a concrete class.

No class should derive from a concrete class.

No method should override an implemented method of any of its base classes.

It is OK to depend on stable classes like String, Integer, JPanel, etc.

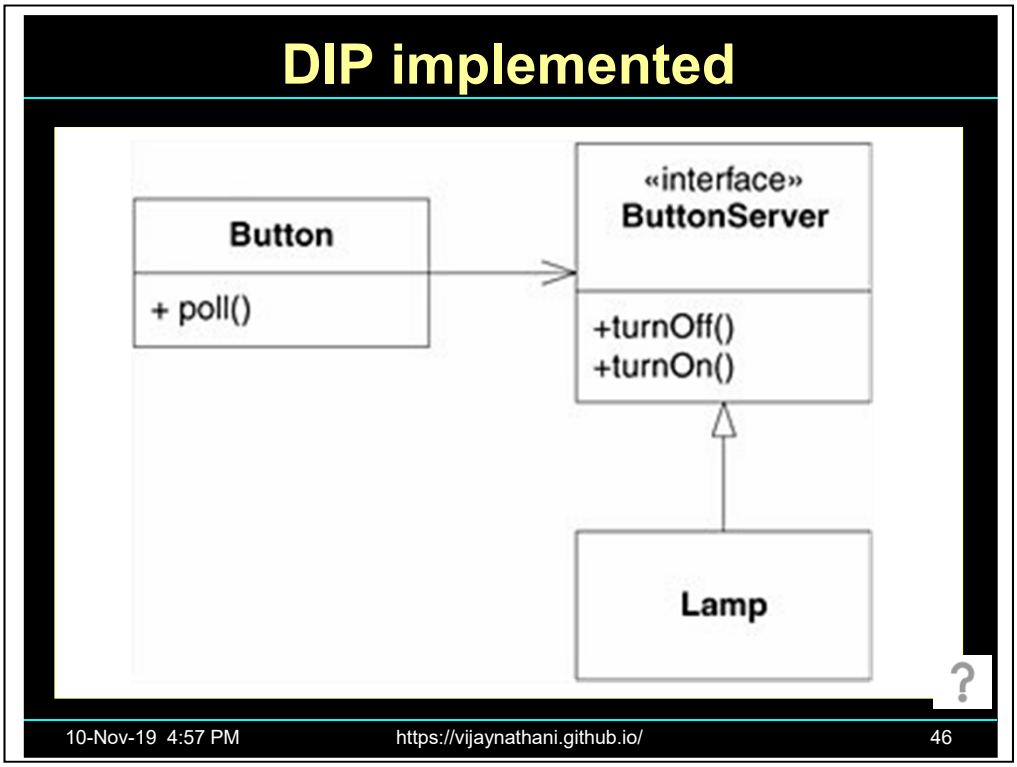
Avoids designs that are rigid, Fragile and Immobile.

Example: We have a classes: Copy, Keyboard, Printer. The Copy reads from keyboard and prints to the printer. If we use DIP, we have an abstraction for Keyboard and Printer. So we can add input and output devices later on.

Example: Collections in Java.

Example: Customer is a class. Employee is a class. Employees are now allowed to purchase on credit. We need an interface Buyer.

For every variable use the maximum abstract type possible.



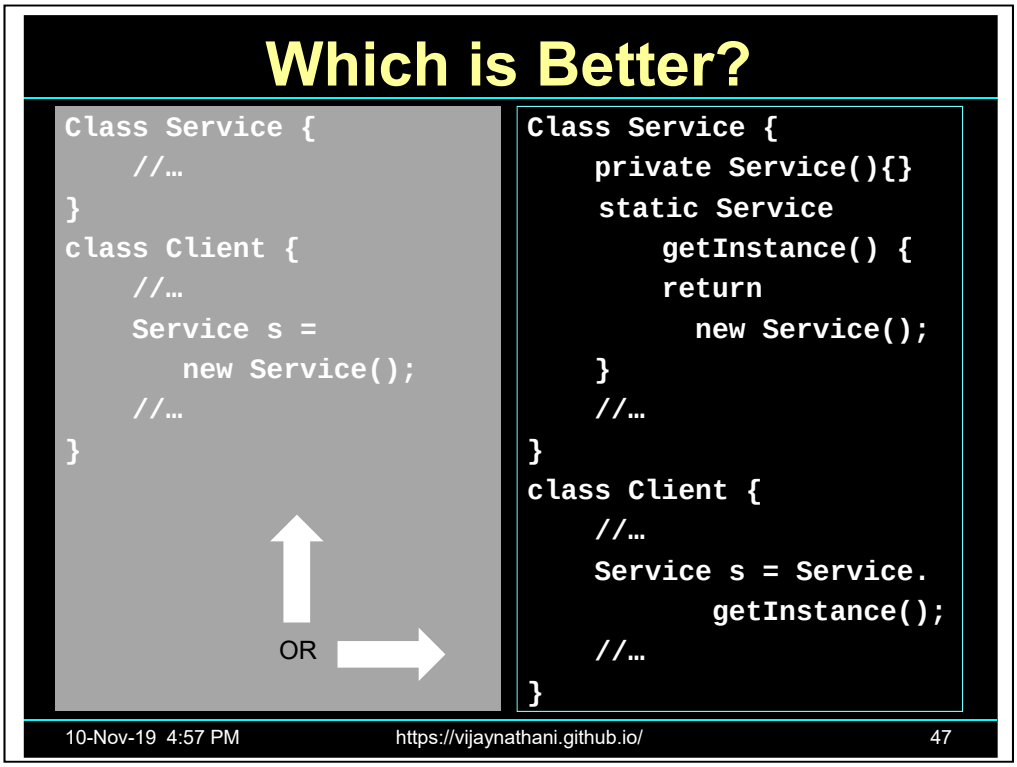
Advantages

- Clients are unaware of the specific class of the object they are using
- One object can be easily replaced by another
- Object connections need not be hardwired to an object of a specific class, thereby increasing flexibility
- Loosens coupling
- Increases likelihood of reuse
- Improves opportunities for composition since contained objects can be of any class that implements a specific interface

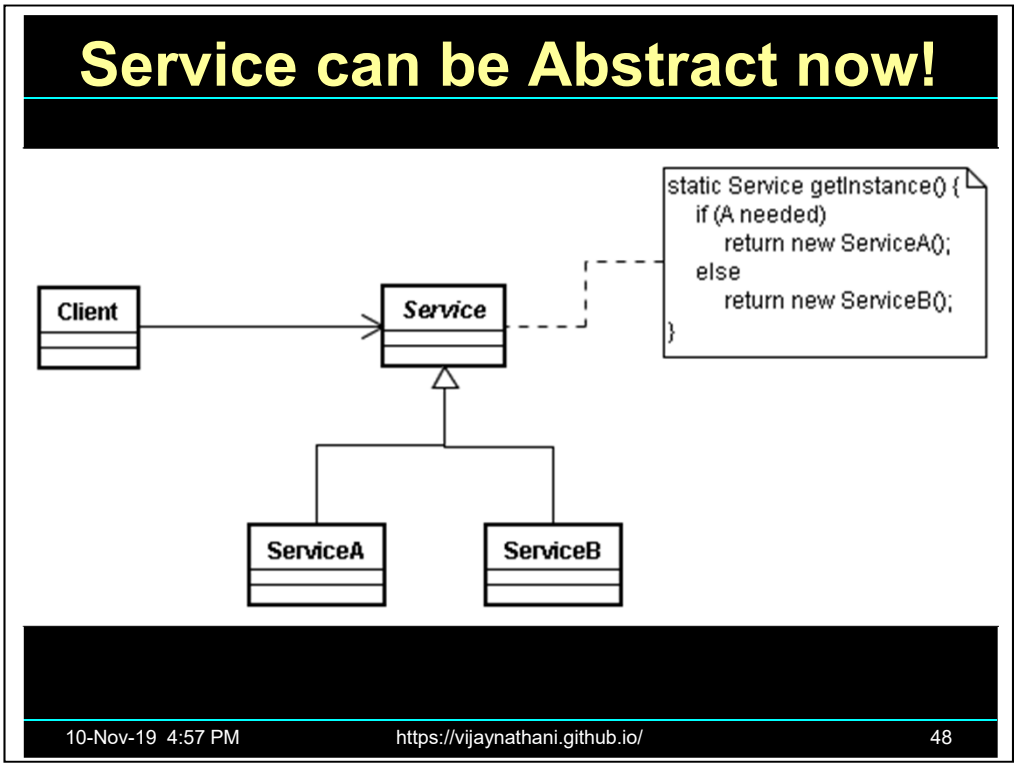
Disadvantages

- Modest increase in design complexity

Q96, Q94, Q93, Q91 - DIP



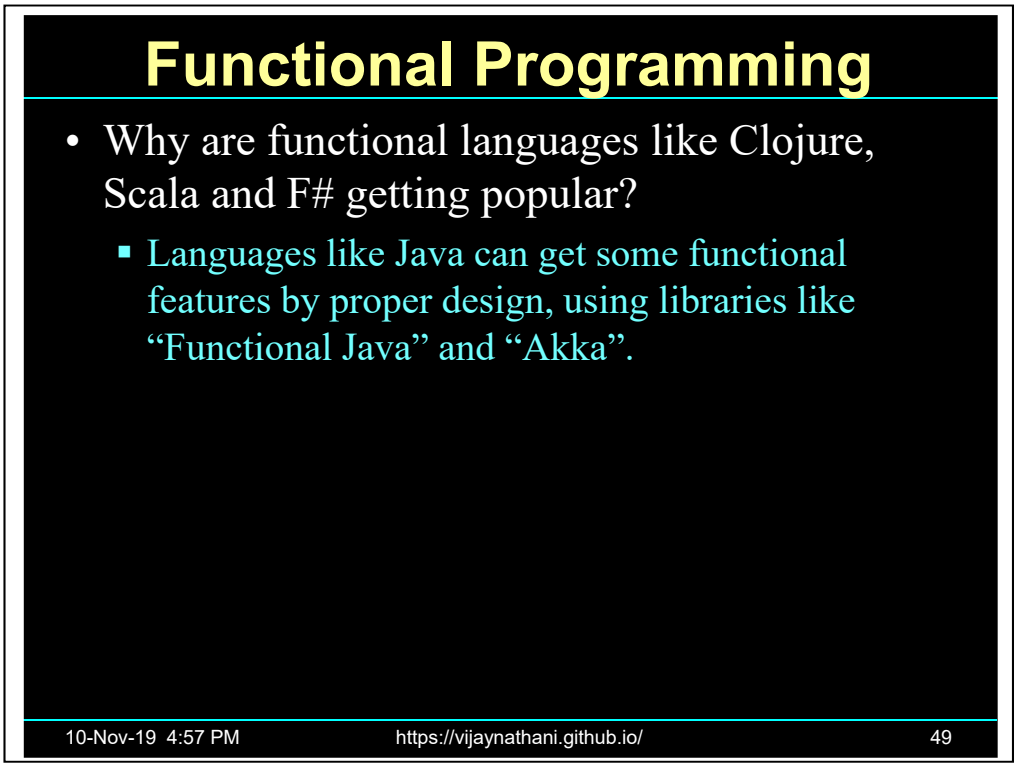
The right side is better. Service class can be made abstract tomorrow.



The main advantages of using Static function for construction instead of constructor directly is:

1. Static methods have a name. This serves as documentation e.g. BigInteger(int, int, Random) is not clear but BigInteger.getProbablePrime(int, int, Random) is more clear.
2. The static method can return the same object for multiple calls e.g. if the object is immutable e.g. Boolean class has two objects TRUE and FALSE. For repeated calls, the same object can be returned.
3. Last, the static function can return a subtype object.
4. Instead of writing “Map<String,Integer>m=new Hashmap<String,Integer>();” we can write

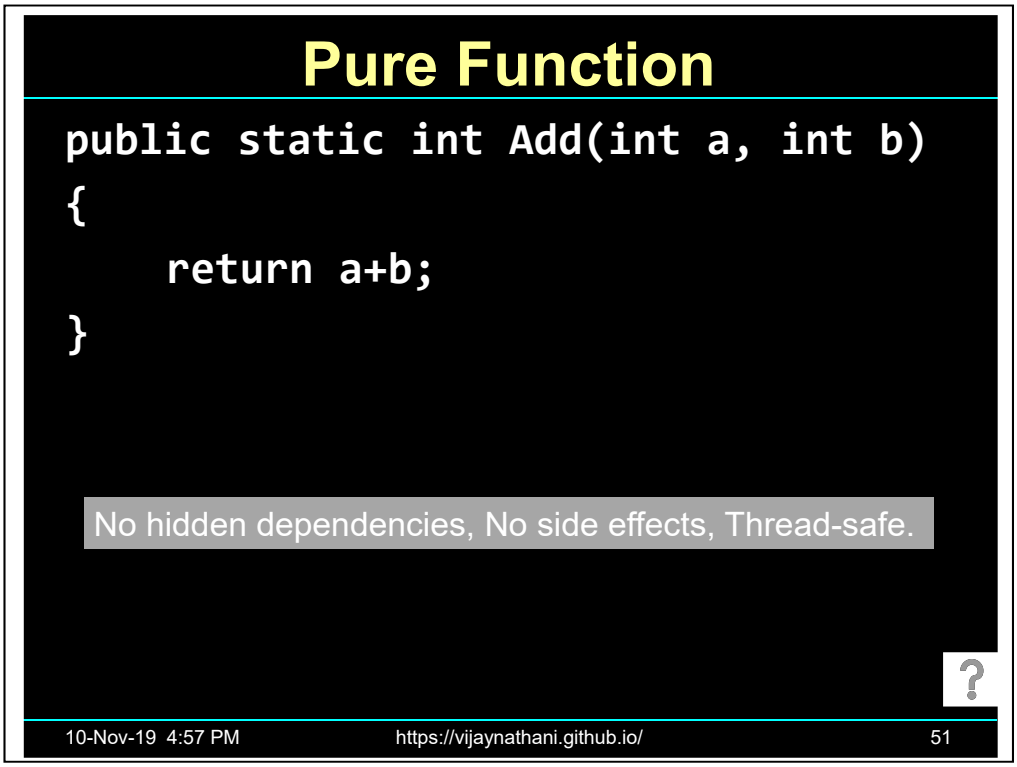
   “Map<String,Integer>m=Hashmap.newInstance();”



Functional language advantages:
functions are first class citizens
functions are side-effect free
declarative. Uses recursion to reduce the size of the problem.

Pure Functions

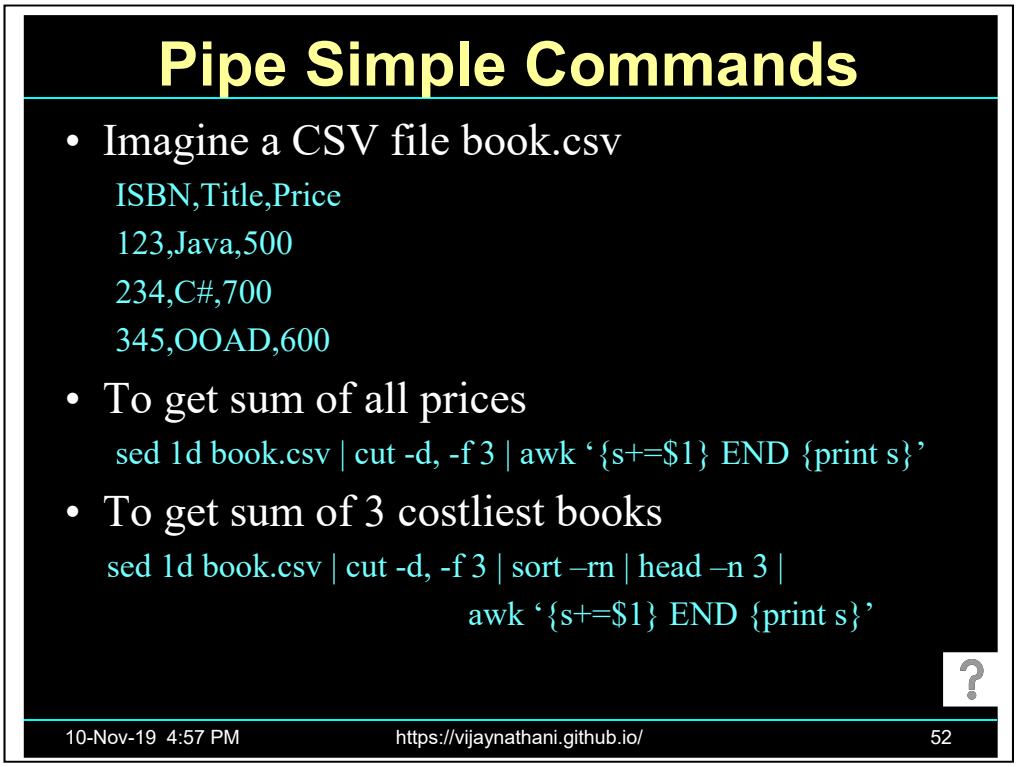
- A function that computes output based on parameters passed only.
  - No access to instance variables or global variables
- The function does not alter any input parameters.



Also known as side-effect free functions.

Q60.

Q61



This is Linux philosophy. Do a complex process by performing simple tasks and then piping them.

Instead of building a huge complex class, build simple classes. Pipe output of one class to another. This will result in more flexible coding. More portions will be reusable.

Q62

Iteration

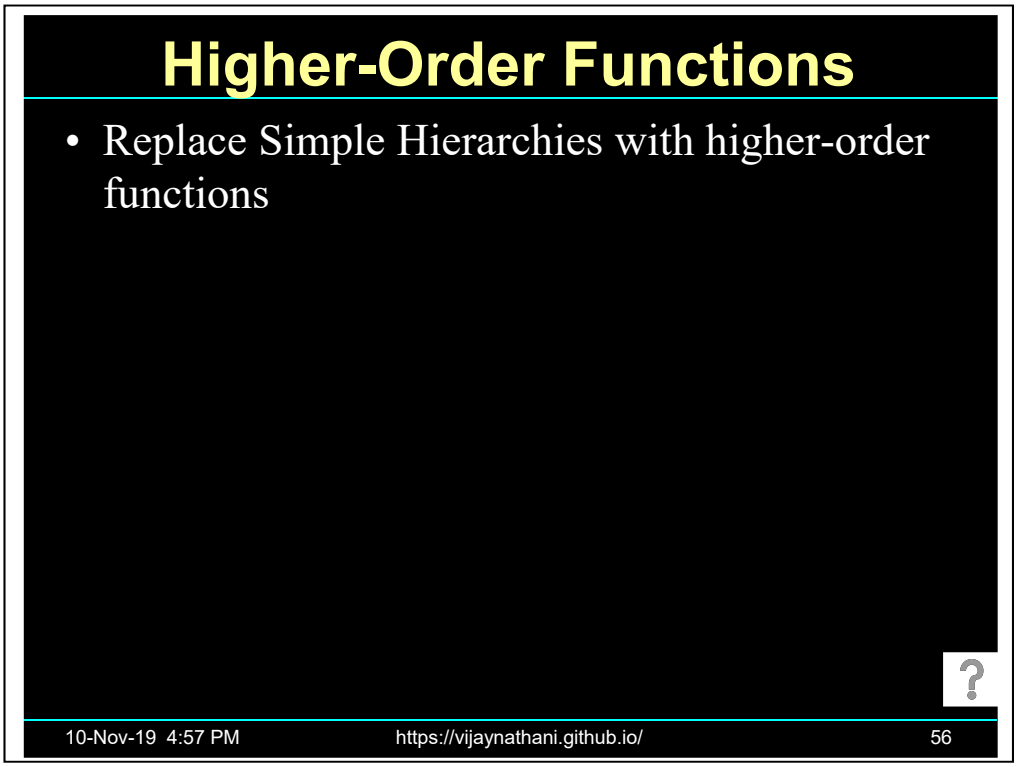
- Functional programming supports Lamda/Closure.
- Prefer internal iteration to external iteration.

Iteration (Java)

```
List<Integer> numbers =
   Arrays.asList(1, 2, 3, 4, 5, 6);
//External Iteration
for (Integer n : numbers)
    System.out.println(n);
//Internal iteration
numbers.stream().forEach(
    System.out::println);
```

Iteration (C#)

```
var numbers = new List<int>
        {1, 2, 3, 4, 5, 6};
//External Iteration
foreach (var e in numbers)
    Console.WriteLine(e);

//Internal iteration
numbers.ForEach(Console.WriteLine);
```



Higher-order functions means a function that either accepts a function or returns a function.

If a hierarchy has only one abstract function, derived class has no instance variables and all classes are small, then replace hierarchy by higher-order function.

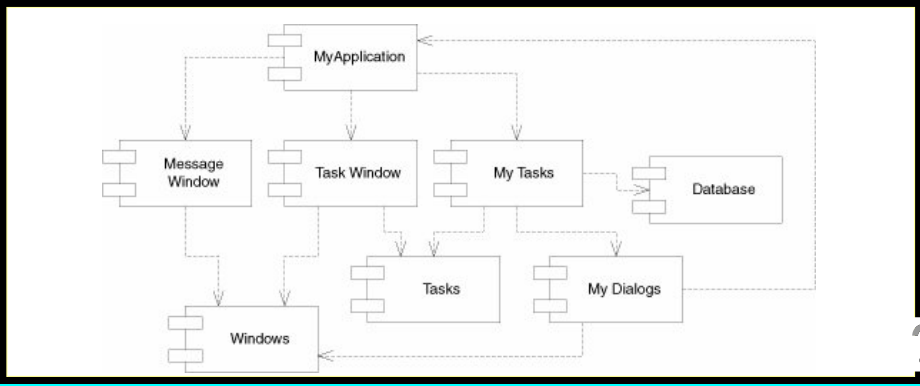
Q77

Principles

- The Release/Reuse Equivalency Principle
  - The granule of reuse is the granule of release.
    - Either all the classes in a component are reusable, or none of them are.
- The Common Reuse Principle
  - The classes in a component are reused together.
    - If we reuse one of the classes in a component, we reuse them all.
- The Common Closure Principle
  - Classes that change together, belong together in a package.

Principles

- Acyclic Dependencies Principles (ADP)
  - No cycles in the package diagram of a particular layer.
  - No cycles in the component dependency graph



10-Nov-19 4:57 PM https://vijaynathani.github.io/ 58

Q80reference – zipengine Q81reference - FileCopier Q82reference - faxNo

A component contains one or more packages. If there are no cycles in package diagram, then there will be no cycles in component diagram.

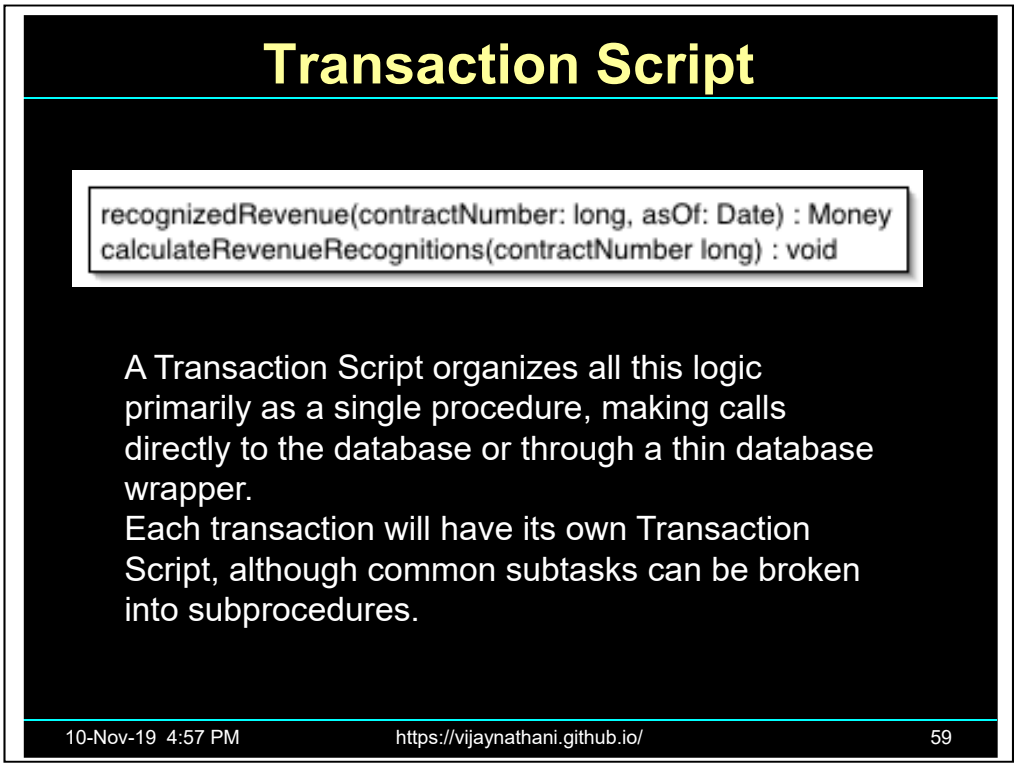
Cycles are introduced in package diagram as an example of code deterioration. So run JDepend tools regularly.

Example of violation

•java.util <-> java.lang : Java.lang.String imports java.util.Locale. java.util.Locale uses String.

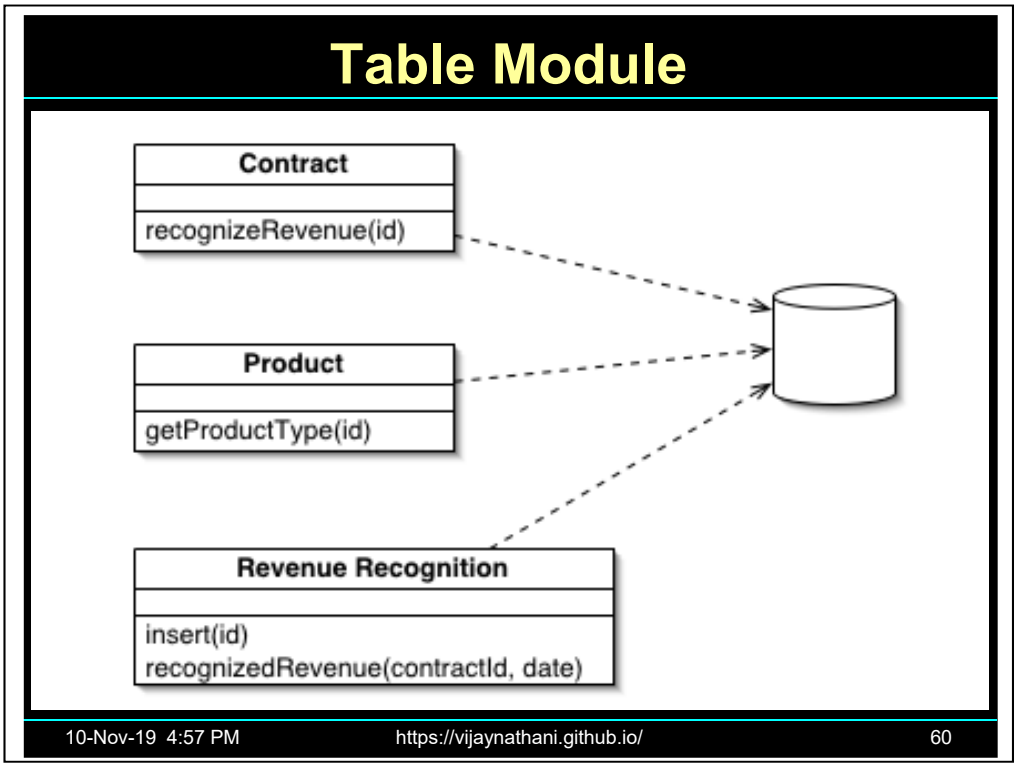
•Hibernate

Good example: Spring does not contain a single package circle.



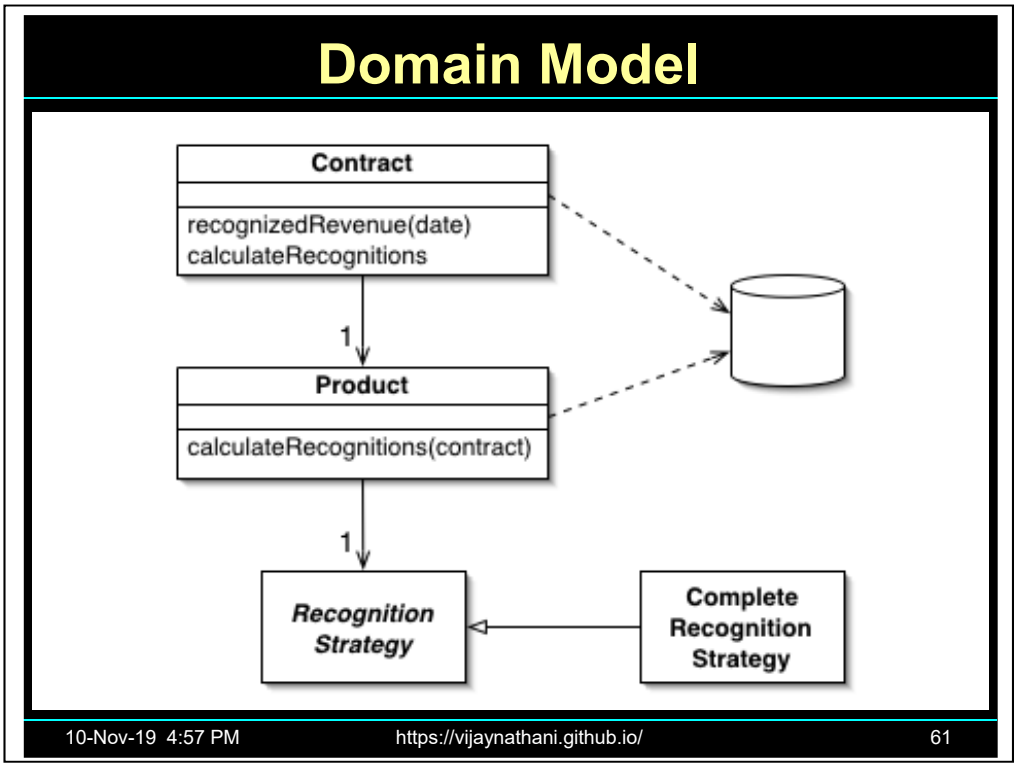
Transaction Script is similar to batch programs in that all functionality is described from start till end in a method. Transaction Script is very simple and useful for simple problems, but breaks down when used for dealing with high complexity. Duplication will be hard to avoid. Still, very many very large systems are built this way. Been there, done that, and I've seen the evidence of duplication even though we tried hard to reduce it. It crops up.

This model is useful for CRUD application and people coming from Data Processing (Cobol) background i.e. People with limit OO skills



A Table Module organizes domain logic with one class per table in the data-base, and a single instance of a class contains the various procedures that will act on the data. The primary distinction with Domain Model is that, if you have many orders, a Domain Model will have one order object per order while a Table Module will have one object to handle all orders.

Table Module encapsulates a Recordset, and then you call methods on the Table Module for getting information about that customer with id 42. To get the name, you call a method and send id 42 as a parameter. The Table Module uses a Recordset internally for answering the request. This certainly has its strengths, especially in environments when you have decent implementations for the Recordset pattern. One problem, though, is that it also has a tendency to introduce duplication between different Table Modules. Another drawback is that you typically can't use polymorphistic solutions to problems because the consumer won't see the object identities at all, only value-based identities. It's a bit like using the relational model instead of the object-oriented model.



A Domain Model creates a web of interconnected objects, where each object represents some meaningful individual, whether as large as a corporation or as small as a single line on an order form.

Domain Model instead uses object orientation for describing the model as close to the chosen abstractions of the domain as possible. It shines when dealing with complexity, both because it makes usage of the full power of object orientation possible and because it is easier to be true to the domain. Usage of the Domain Model pattern isn't without problems, of course. A typical one is the steep learning curve for being able to use it effectively.

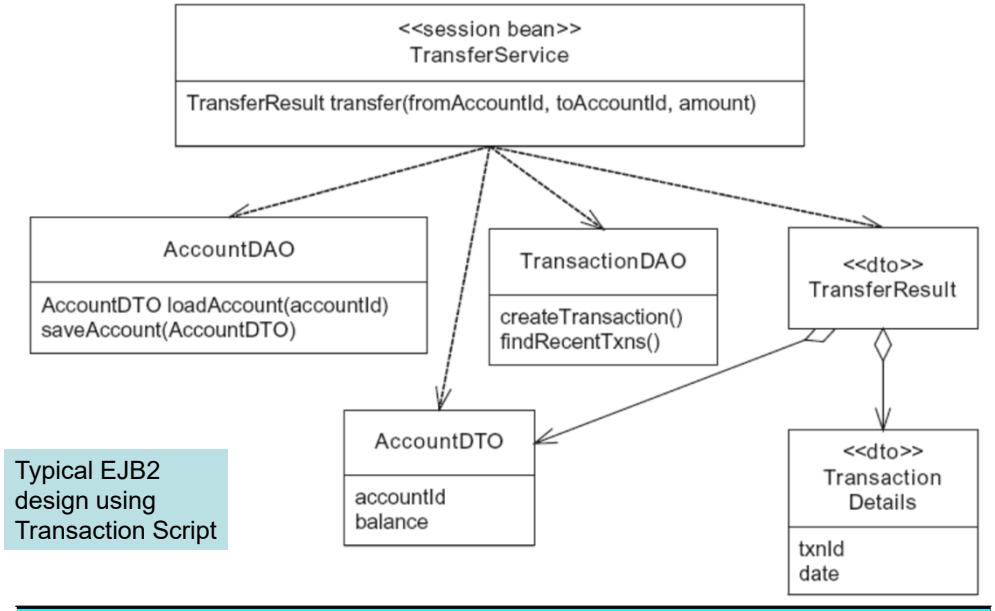
Domain Model is independent of whether we are using J2EE, Spring+Hiberate, .NET, Swing, C++ or windows rich client application.

Domain Model should enforce behavior and rules. It is usually developed in an iterative fashion.

Example UI

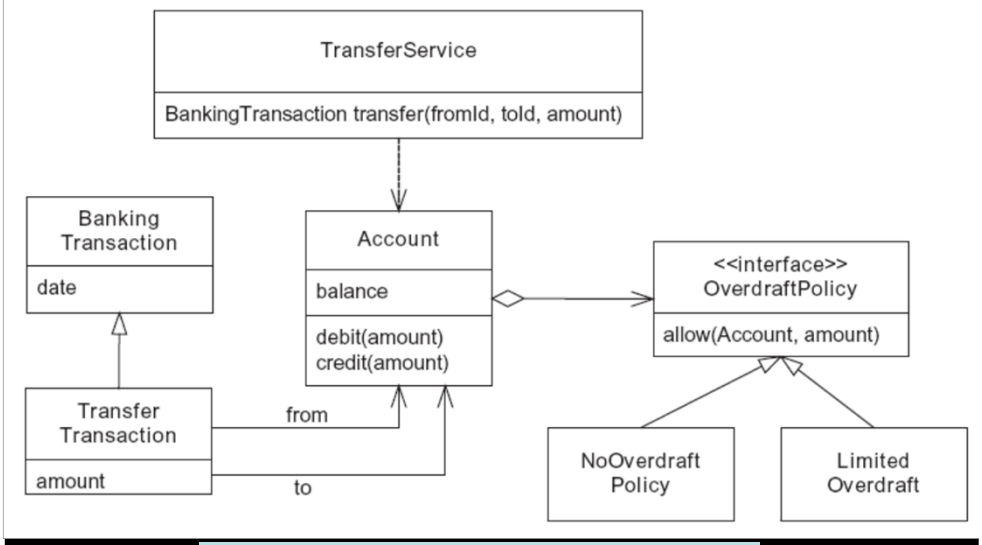
Example banking application

Money Transfer



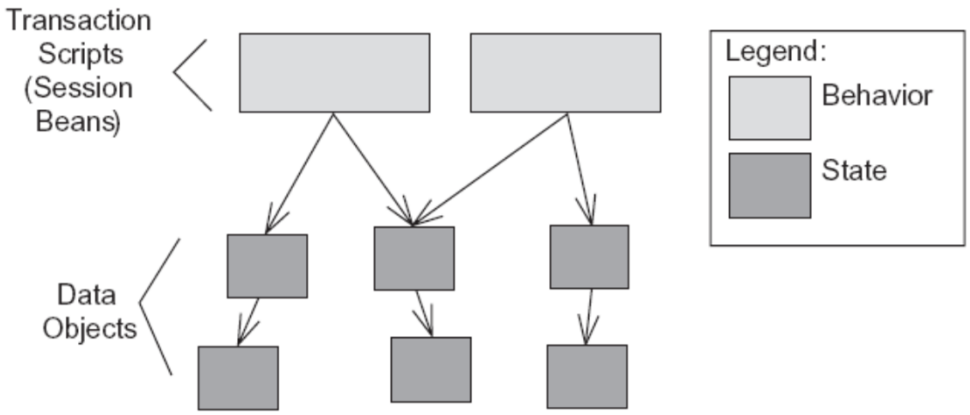
Typical EJB2 design using Transaction Script

Domain Model

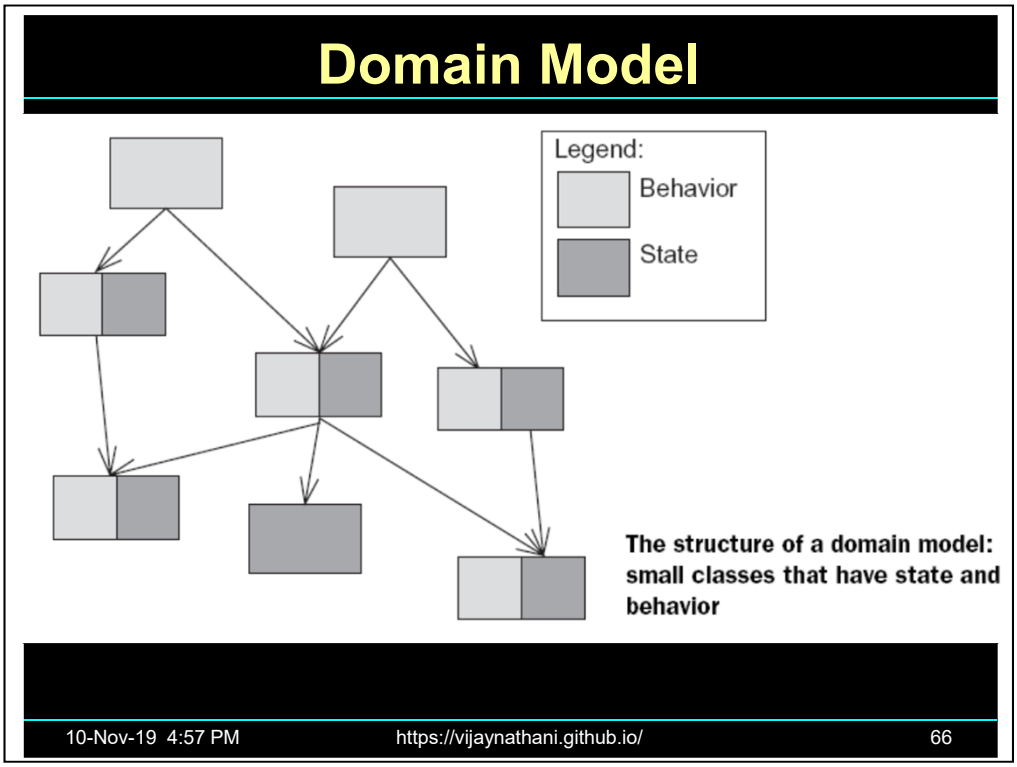


Transferring Money from one account to another

Transaction Script



The structure of procedural design: large transaction script classes and many small data objects



Helps in capturing domain expertise

Structures application code along the lines of business processes

- Through object-oriented design

Persistence is in second place

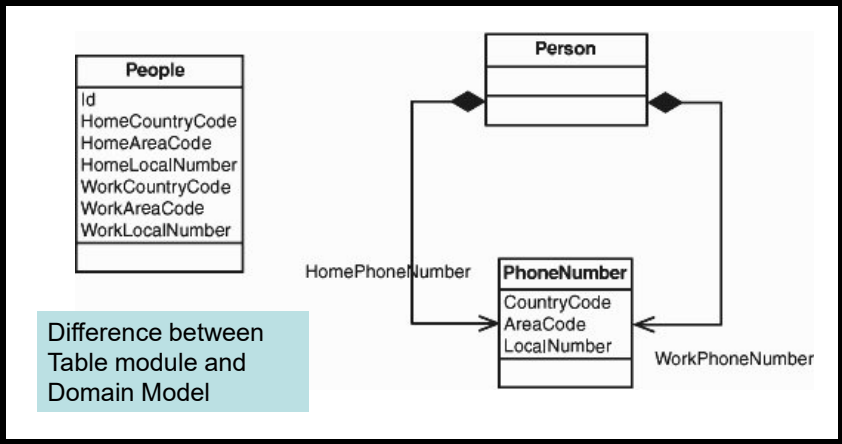
Can handle complexity

Need

- investment
- strong design and technical skills
- Regular access to domain experts

Disadvantages of Table Module

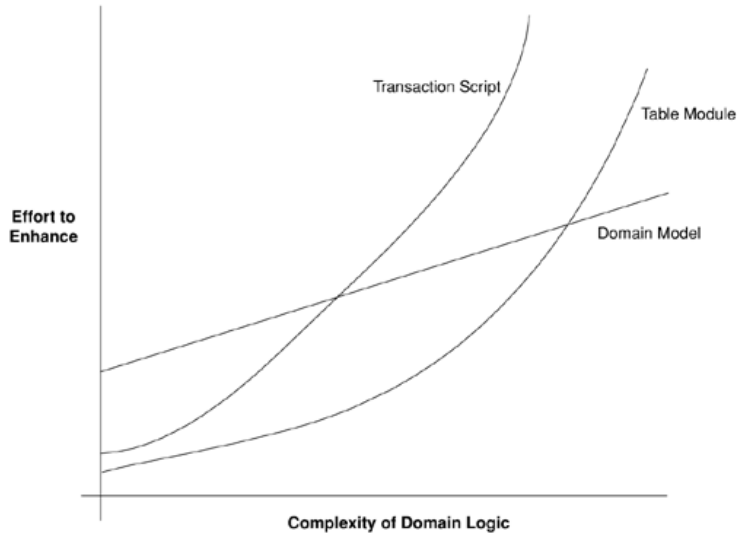
- Not cost effective
- No competitive advantage



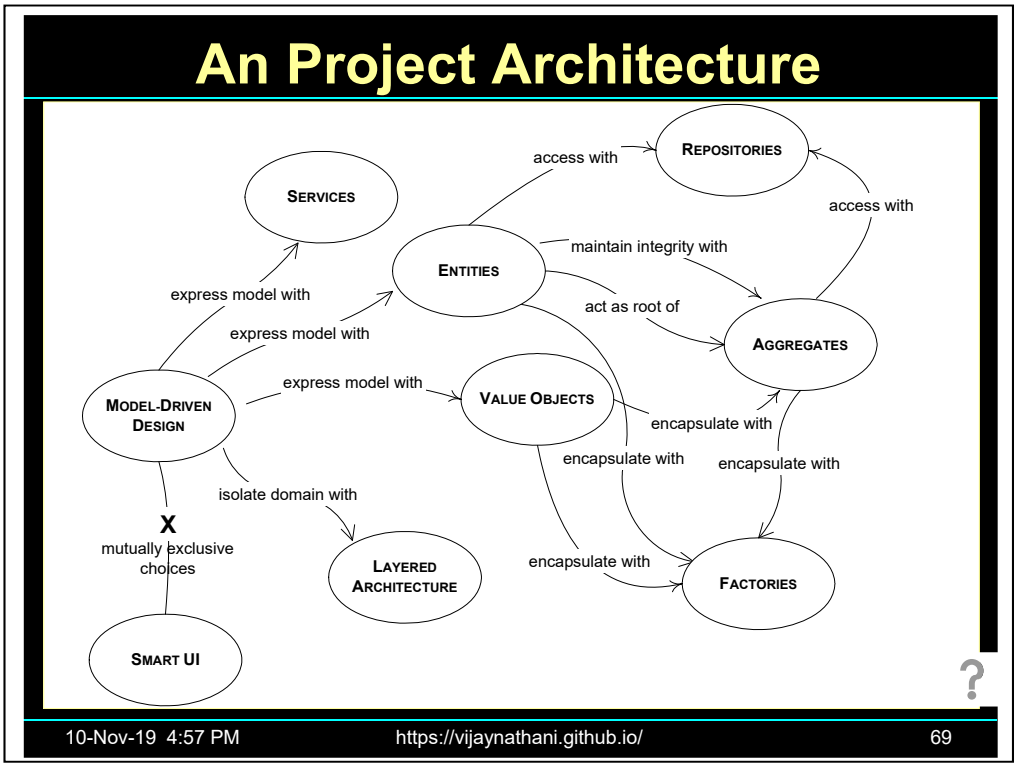
Difference between Table module and Domain Model

Deciding whether or not to use the Domain Model pattern is actually an up-front architecture design decision, and a very important one because it will affect a lot of your upcoming work.

Domain Logic Patterns



Q90 - layering



QRestaurant

Roles in a domain model:

Entity – Objects with an distinct identity. These are different from J2EE Entity Beans. Try to build Entities only out of value objects.

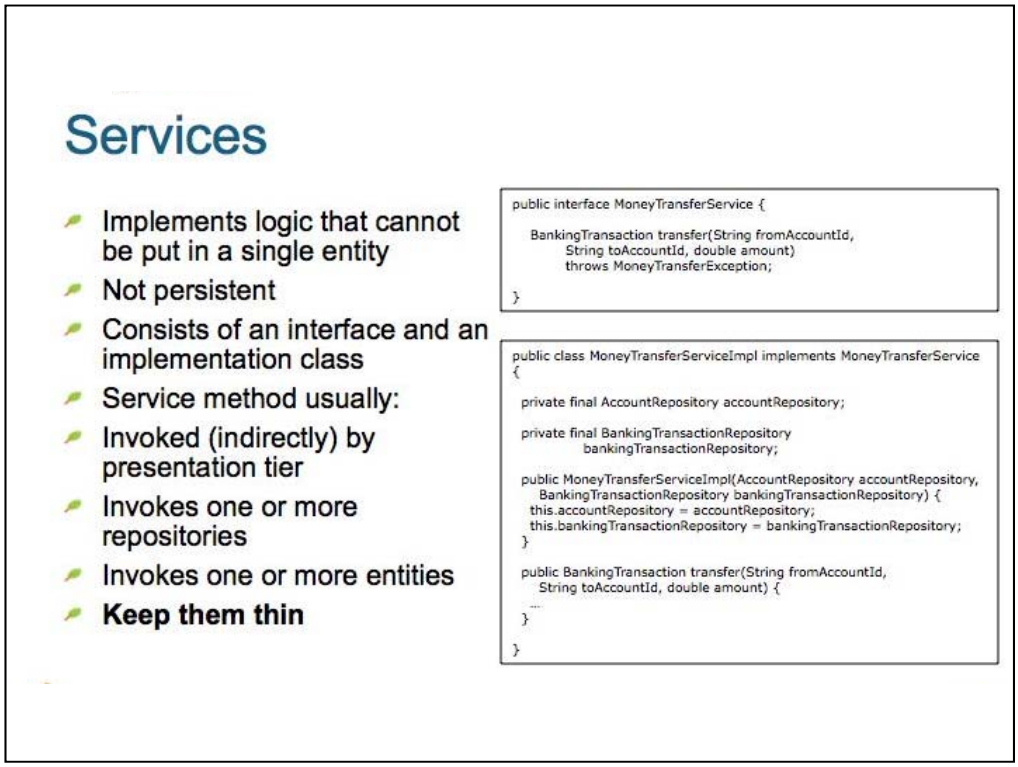
Value – Objects with no distinct identity. These are different from J2EE/Sun Value objects.

Factories – Responsible for creation of Entities.

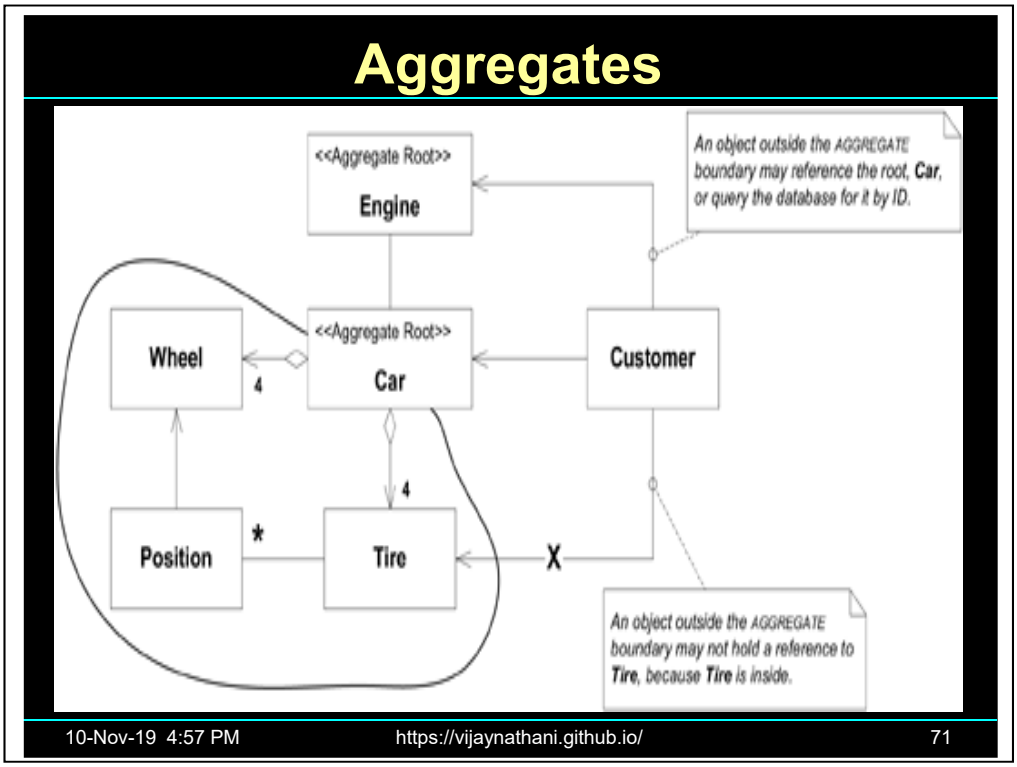
Repositories – Manage collection of entities and encapsulate the persistence mechanism

Services – Implement responsibilities that can’t be assigned to a single class and encapsulate the domain model. They contain an operation offered as a stateless service.

Repositories – Similar to J2EE DAO. Don’t query for value objects, since they have no identity.



Used for transactions, Logging, etc. Aspects used here.



A cluster of related entities and values.

Behaves as an unit.

Has a root.

Has a Boundary.

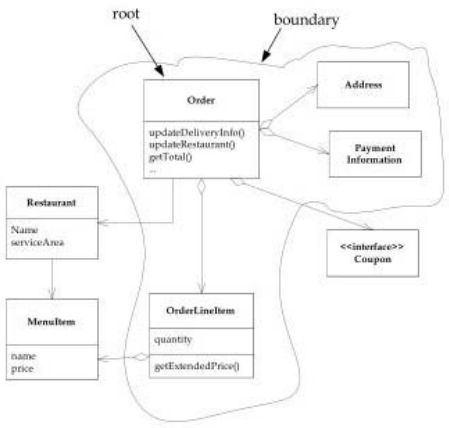
Objects outside the aggregate can only reference the root

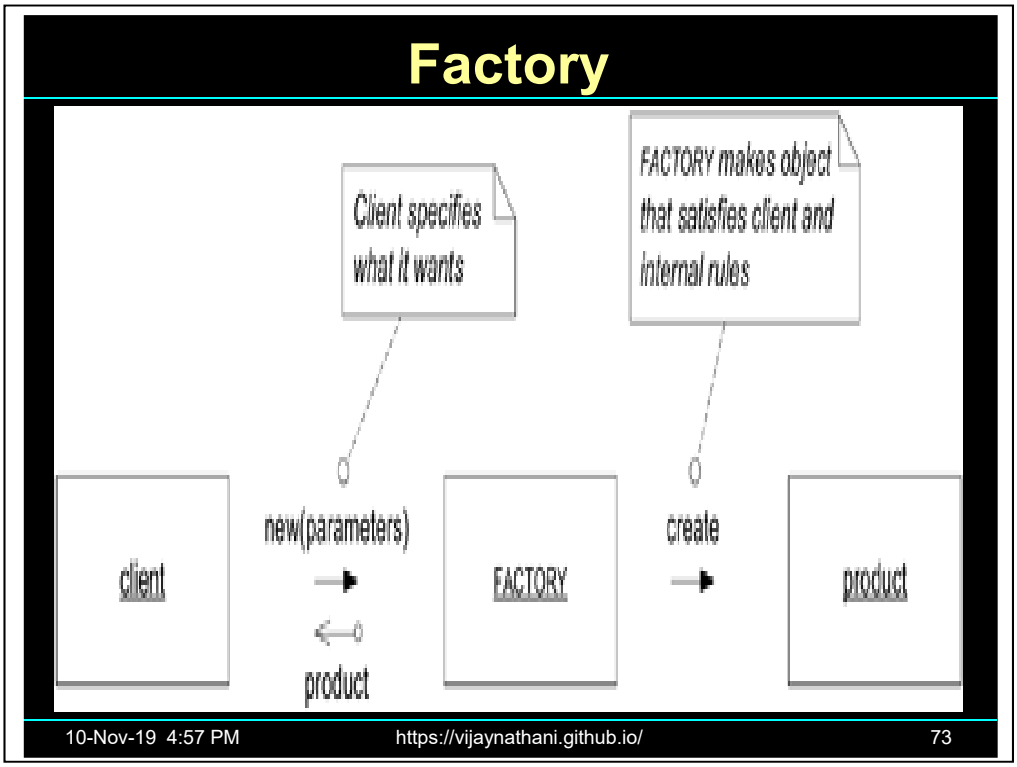
Deleting the root removes everything.

Aggregates provide consistency and transactional boundary.

Aggregates

- A cluster of related entities and values
- Behaves as a unit
- Has a root
- Has a boundary
- Objects outside the aggregate can only reference the root
- Deleting the root removes everything

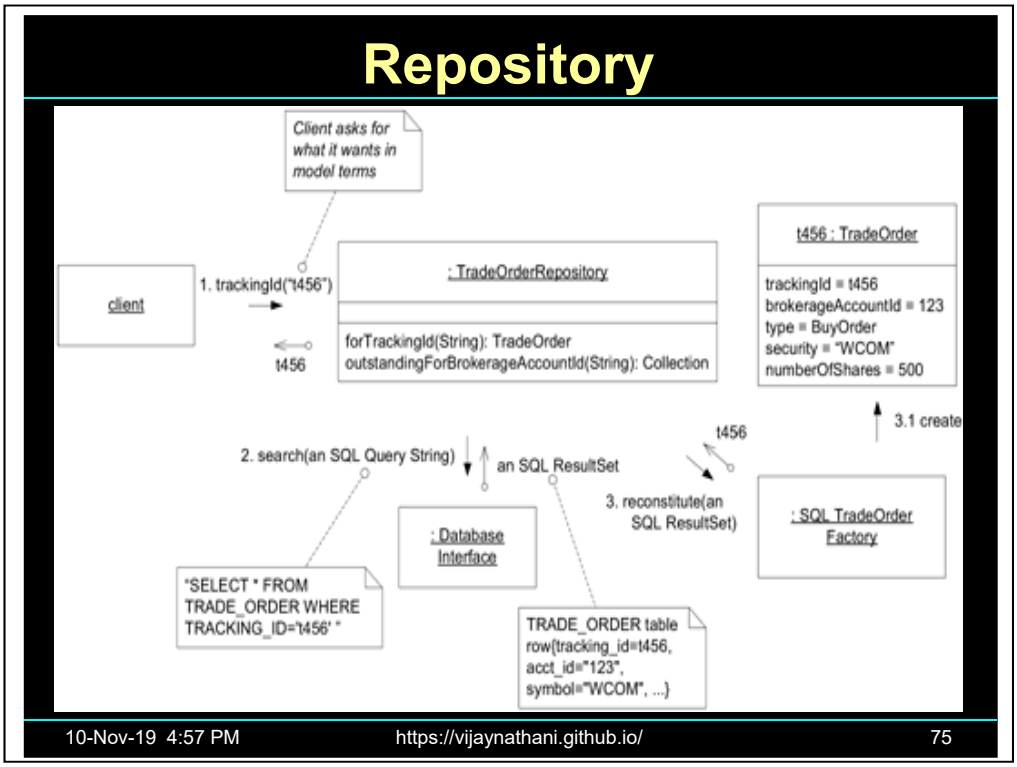




Constructor's should be dead simple. Otherwise, use a factory. E.g. thread pool of Java, sessionfactory of Hibernate

Factories

- Use when a constructor is insufficient
  - Encapsulates complex object creation logic
  - Varying products
- Different kinds of factories
  - Factory classes
  - Factory methods
- Example: OrderFactory
  - Creates Order from a shopping cart
  - Adds line items



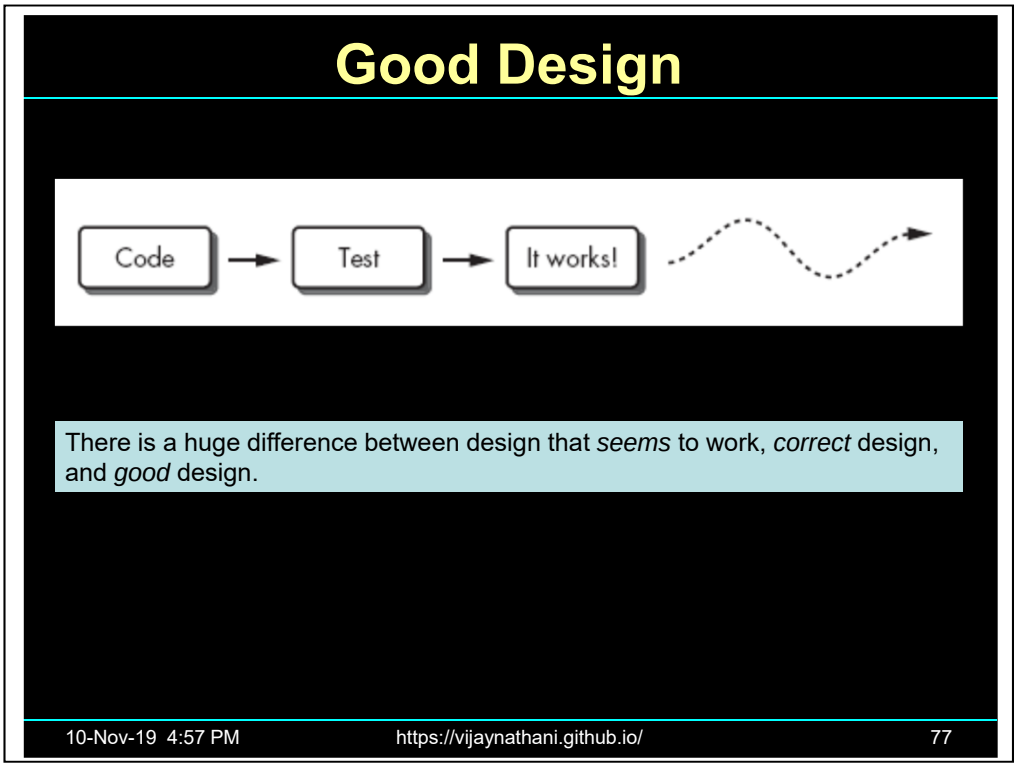
The FACTORY makes new objects; the REPOSITORY finds old objects. The client of a REPOSITORY should be given the illusion that the objects are in memory.

Repositories

- Manages a collection of objects
- Provides methods for:
- Adding an object
- Finding object or objects
- Deleting objects
- Consists of an interface and an implementation class
- Encapsulates database access mechanism
- Keeps the ORM framework out of the domain model
- Similar to a DAO

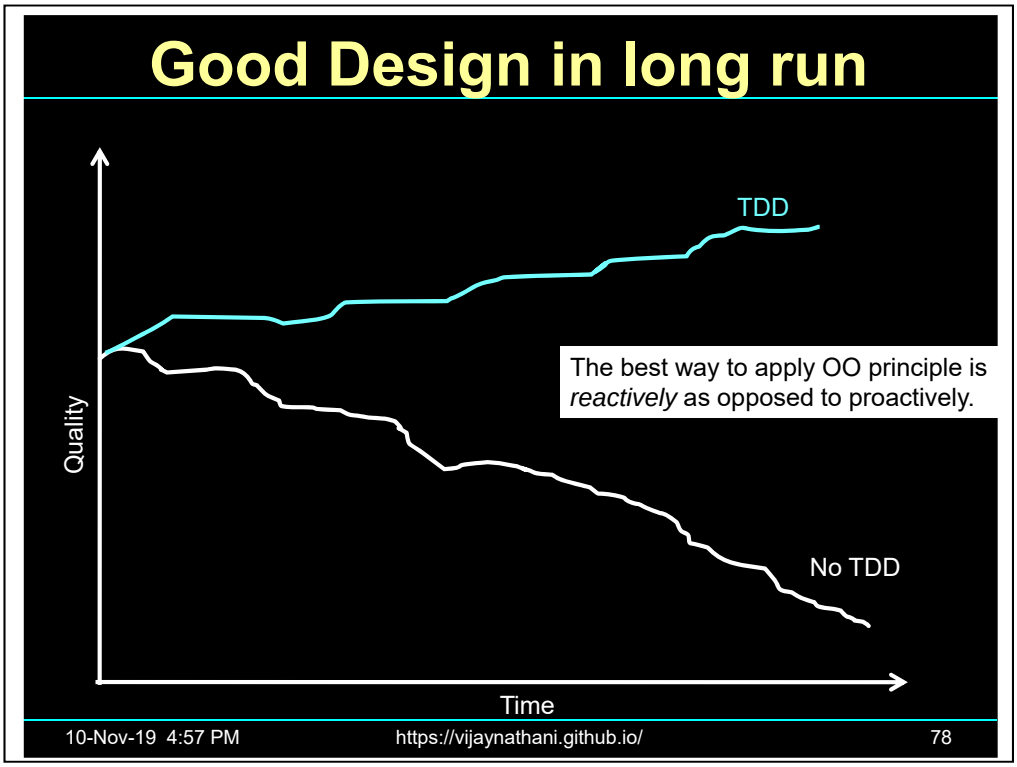
```
public interface AccountRepository {

    Account findAccount(String accountId);

    void addAccount(Account account);

}
```

```
public class HibernateAccountRepository implements AccountRepository {

  private HibernateTemplate hibernateTemplate;

  public HibernateAccountRepository(HibernateTemplate template) {
    hibernateTemplate = template;
  }

  public void addAccount(Account account) {
    hibernateTemplate.save(account);
  }

  public Account findAccount(final String accountId) {
    return (Account) DataAccessUtils.uniqueResult(hibernateTemplate
        .findByNamedQueryAndNamedParam(
            "Account.findAccountByAccountId", "accountId",
            accountId));
  }

}
```



. Defensive programming is a method of prevention.

Debugging is all about finding a cure.

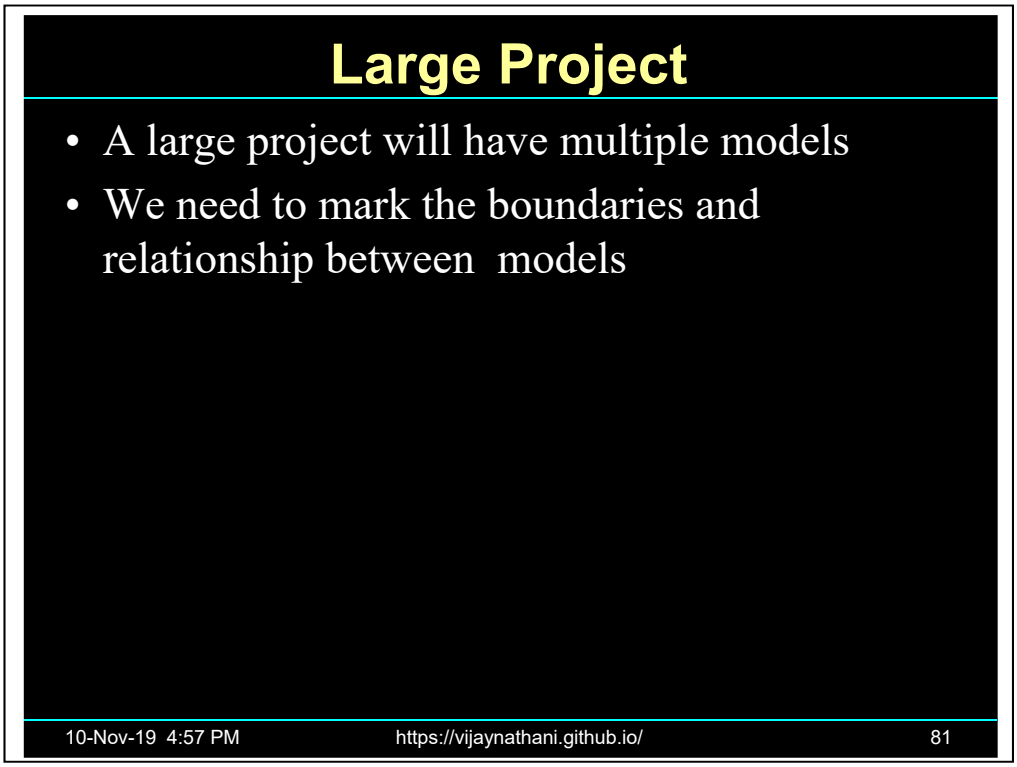


The best way to apply OO principle is *reactively* as opposed to proactively. When you first detect that there is a structural problem with the code, or when you first realize that a module is being impacted by changes in another, *then* you should see whether one or more of these principles can be brought to bear to address the problem

Continuous Integration

Why Model?

- The critical complexity of most software projects is in understanding the domain itself.
- **Model:** A *system of abstractions* that describes *selected* aspects of a domain and can be *used* to solve problems related to that domain.



Multiple models are in play on any large project. Yet when code based on distinct models is combined, software becomes buggy, unreliable, and difficult to understand. Communication among team members becomes confused. It is often unclear in what context a model should not be applied.

A large project will have multiple models. Combining code based on different models is error-prone and hard to understand.

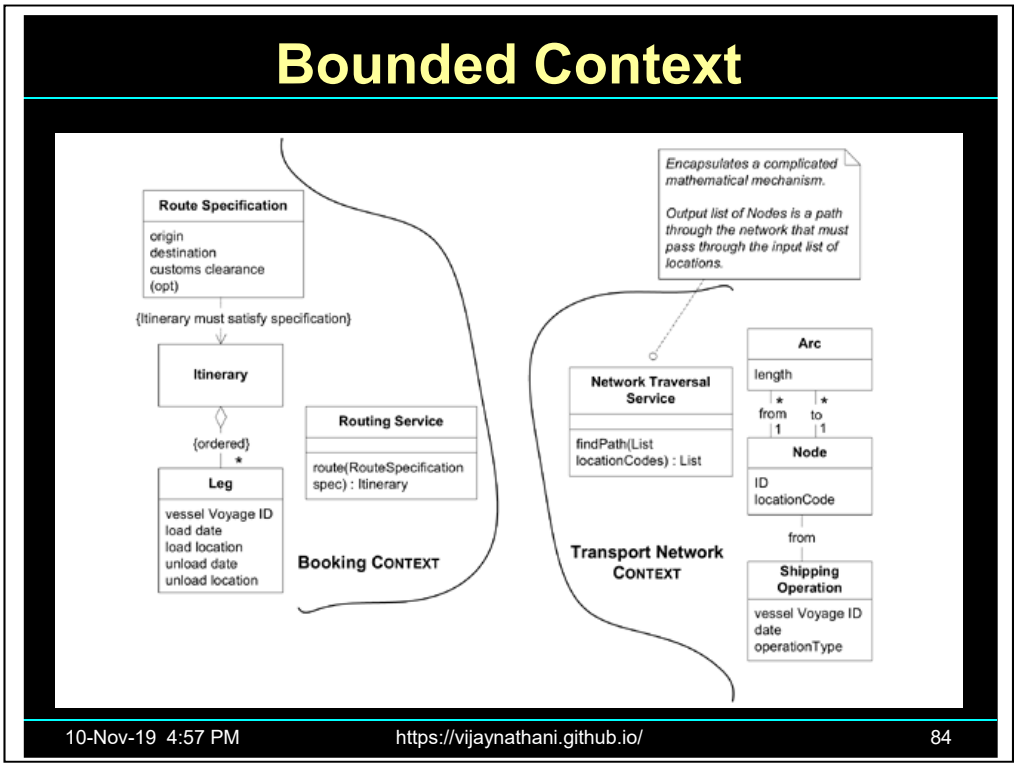
Explicitely define the context within which a model applies. Set boundaries in terms of team organization, usage within specific parts of the application, and physical manifestations like code and database schemas.

Ubiquitous Language

- A good set of composed classes can swallow a lot of business logic and complexity.
  - One word name for important classes / interfaces
    - e.g. Scheduler

Ubiquitous Language

- Examples of composed objects in a financial Application.
  - Money = Amount + Currency
  - CurrencyRate = Amount + Currency + Currency
  - TimeInterval = TimePoint + TimePoint
  - CurrencyQuote = CurrencyRate + TimeInterval



Defiance the context within which a model applies.

Explicitly set boundaries in terms usage within specific parts of the application.

Keep the model strictly consistent within these bounds. Don’t be distracted by issues outside the model.

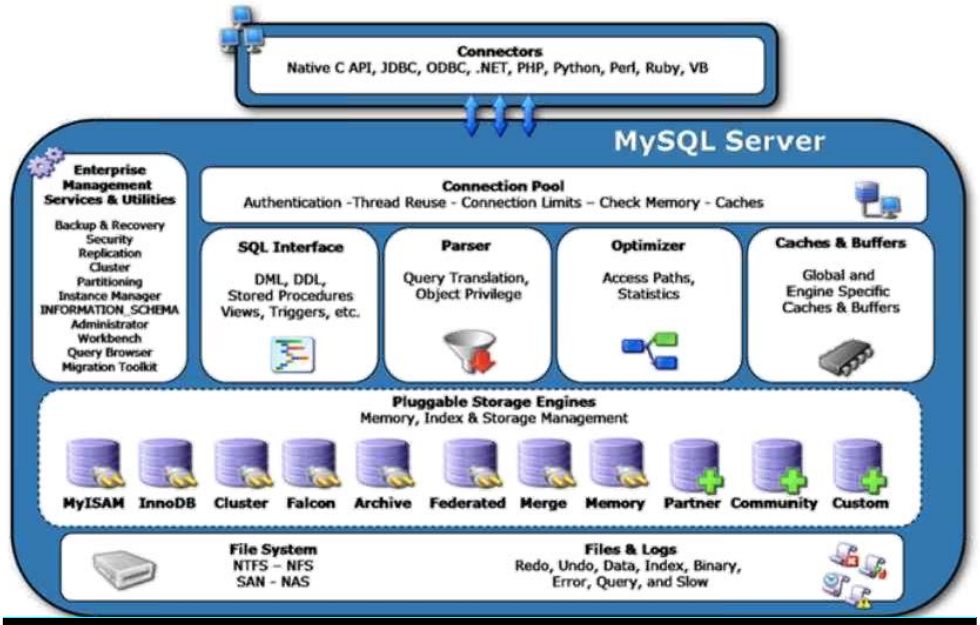
Bounded Context are not packages.

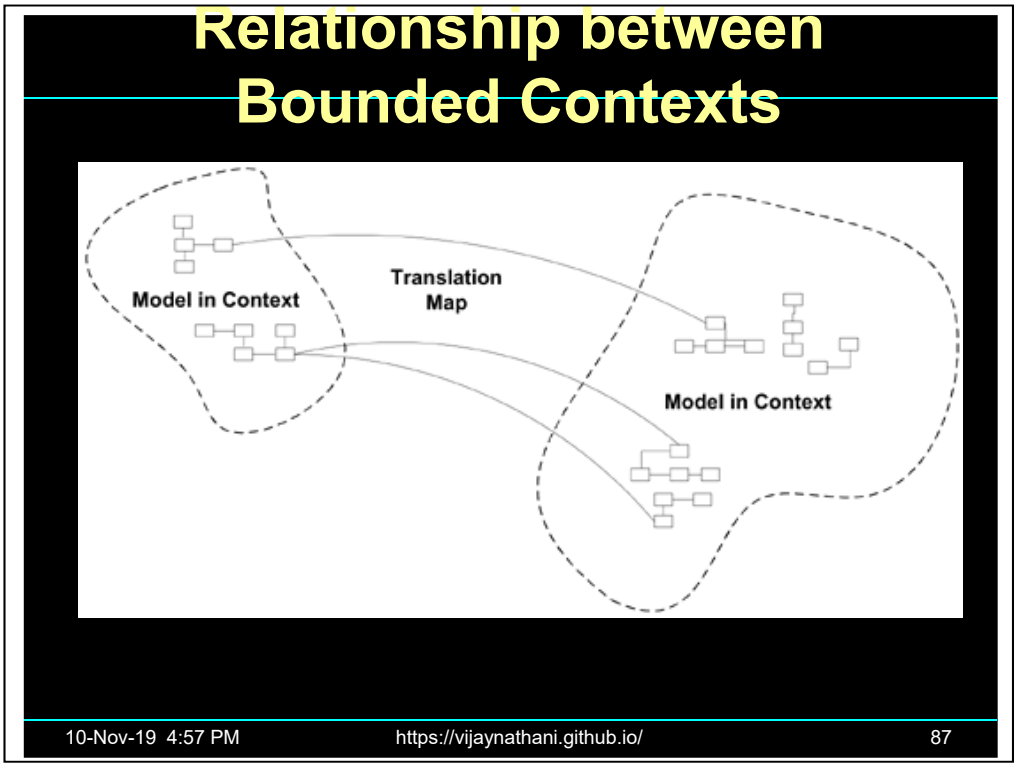
So, what has been gained by defining this BOUNDED CONTEXT? For the teams working in CONTEXT: clarity. Those two teams know they must stay consistent with one model. They make design decisions in that knowledge and watch for fractures. For the teams outside: freedom. They don't have to walk in the gray zone, not using the same model, yet somehow feeling they should.

Toy Bank bounded contexts

- Money - currency
  - Conversion
- Accounting
- Customer
- Loans
  - Origination, Servicing, Collections
- Transfers
- Money Market

MySQL Architecture



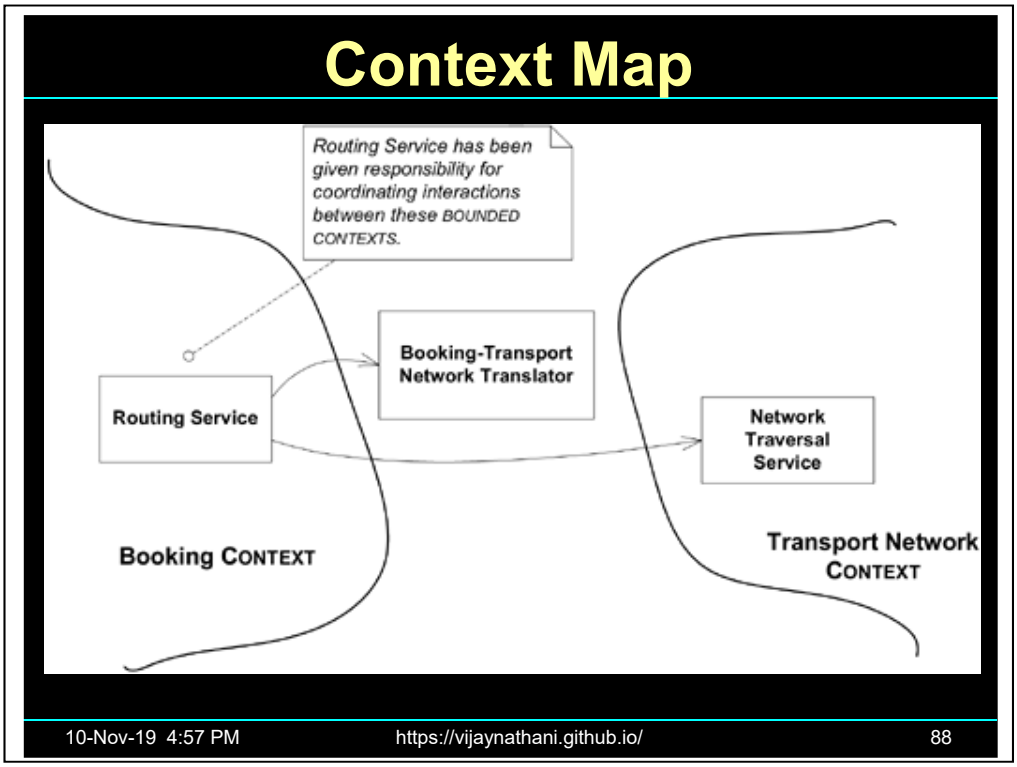


Context Map: Name each Bounded Context.

Describe the points of contact between the models

- communication (requires translation)
- sharing (makes assumptions)

Different bounded contexts usually are in different modules, have different development teams, have their own tests,



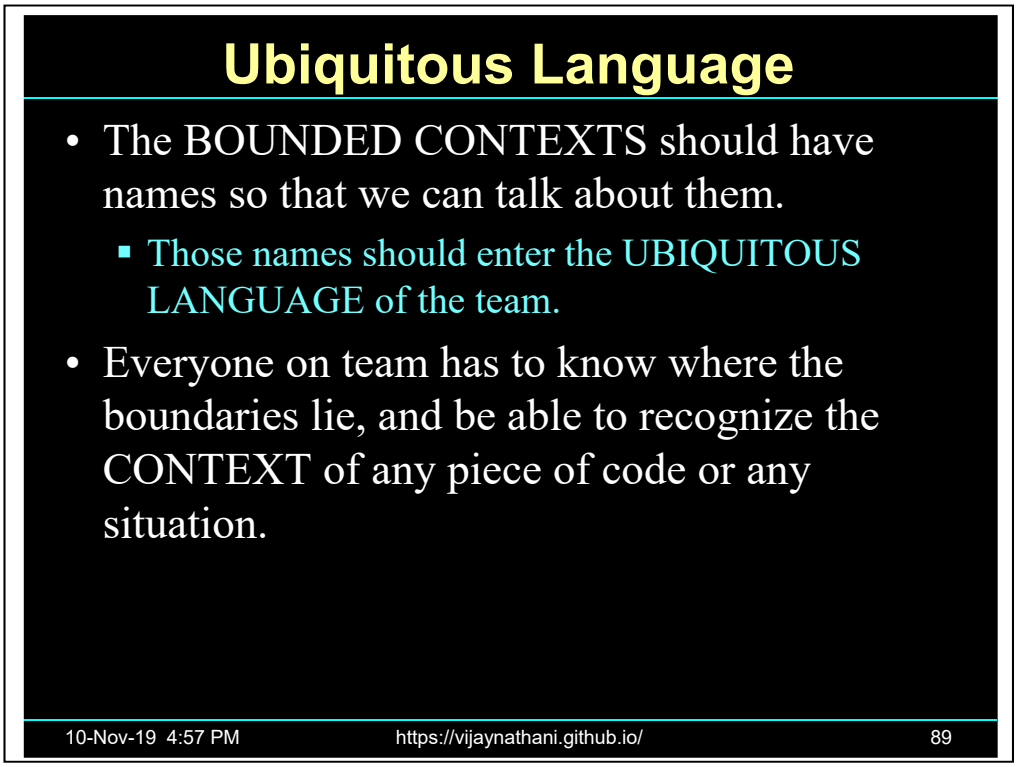
People on other teams won't be very aware of the CONTEXT bounds and will unknowingly make changes that blur the edges or complicate the interconnections. When connections must be made between different contexts, they tend to bleed into each other.

Identify each model in play on the project and define its BOUNDED CONTEXT. This includes the implicit models of non-object-oriented subsystems. Name each BOUNDED CONTEXT, and make the names part of the UBIQUITOUS LANGUAGE.

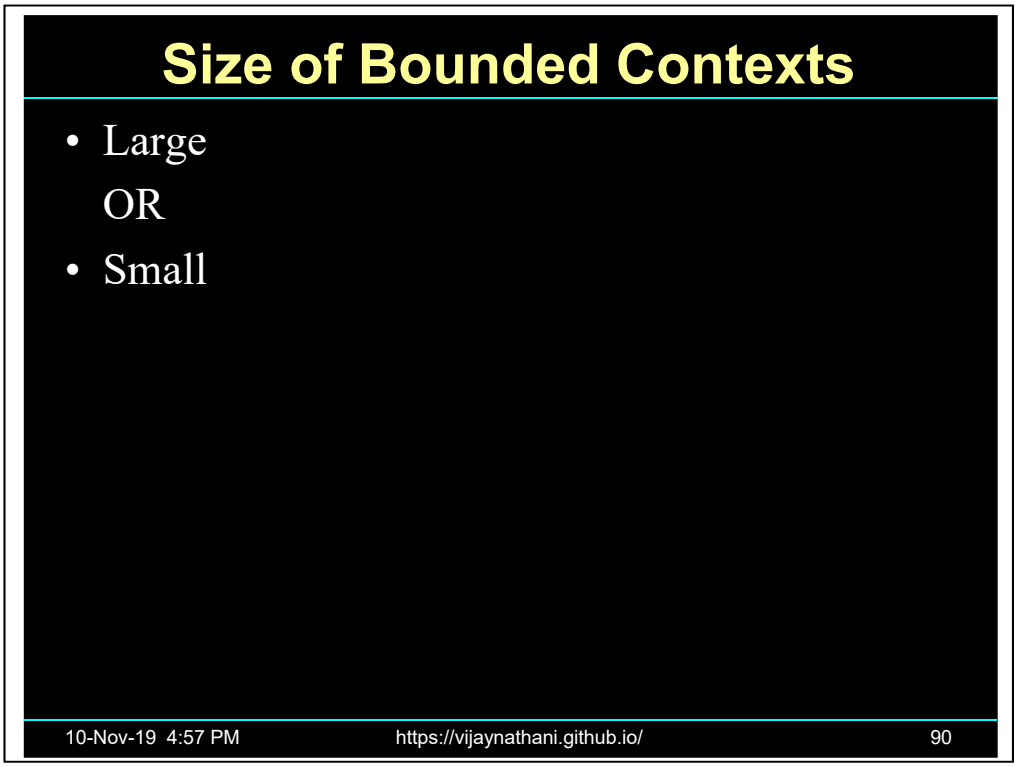
Describe the points of contact between the models, outlining explicit translation for any communication and highlighting any sharing.

Map what is and not what should be.

One Bounded context should have a single unified Model.



In the 12 practices of Extreme Programming, the role of a SYSTEM METAPHOR could be fulfilled by a UBIQUITOUS LANGUAGE.



Favoring Larger BOUNDED CONTEXTS

Flow between user tasks is smoother when more is handled with a unified model.

It is easier to understand one coherent model than two distinct ones plus mappings.

Translation between two models can be difficult (sometimes impossible).

Shared language fosters clear team communication.

Favoring Smaller BOUNDED CONTEXTS

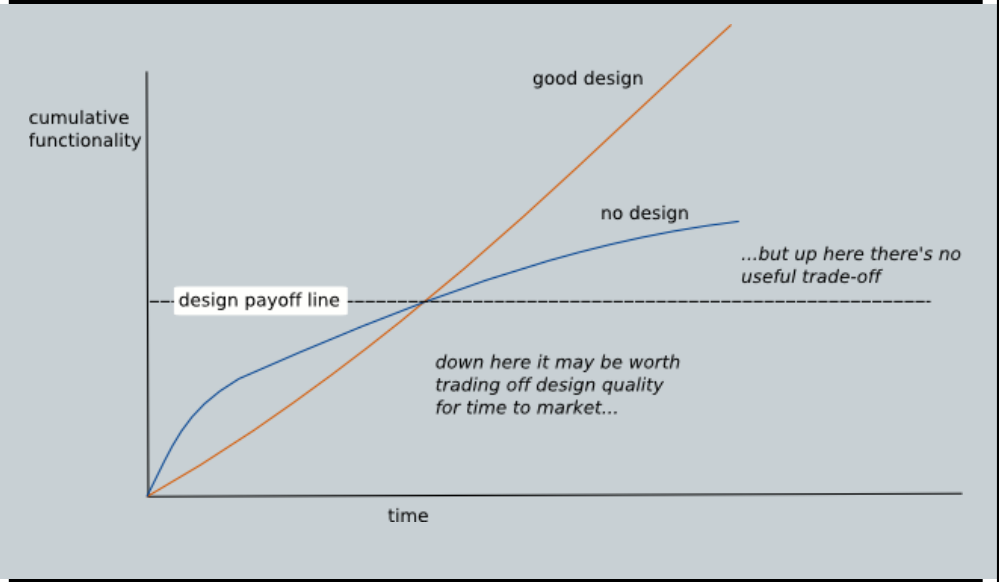
Communication overhead between developers is reduced.

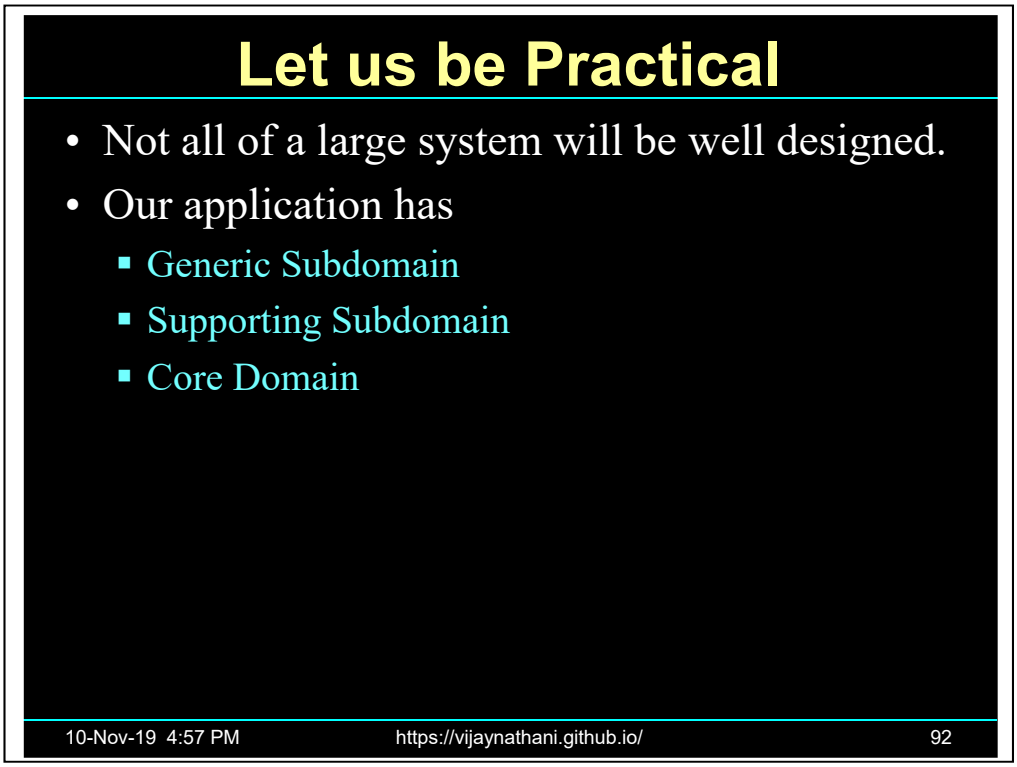
CONTINUOUS INTEGRATION is easier with smaller teams and code bases.

Larger contexts may call for more versatile abstract models, requiring skills that are in short supply.

Different models can cater to special needs or encompass the jargon of specialized groups of users, along with specialized dialects of the UBIQUITOUS LANGUAGE.

Is Good Design worth it?





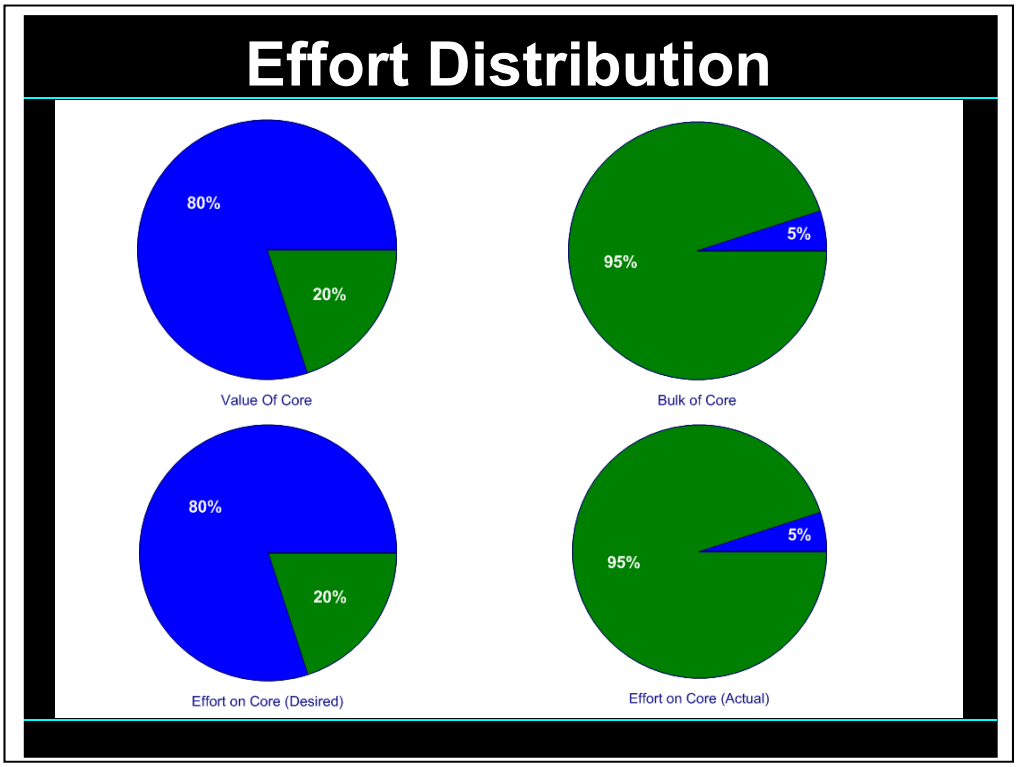
Core domain:

The core domain is the one with business value, the one that makes your business unique. Apply your best talent to the core domain. Spend a lot of time on it and make it as good as you can.

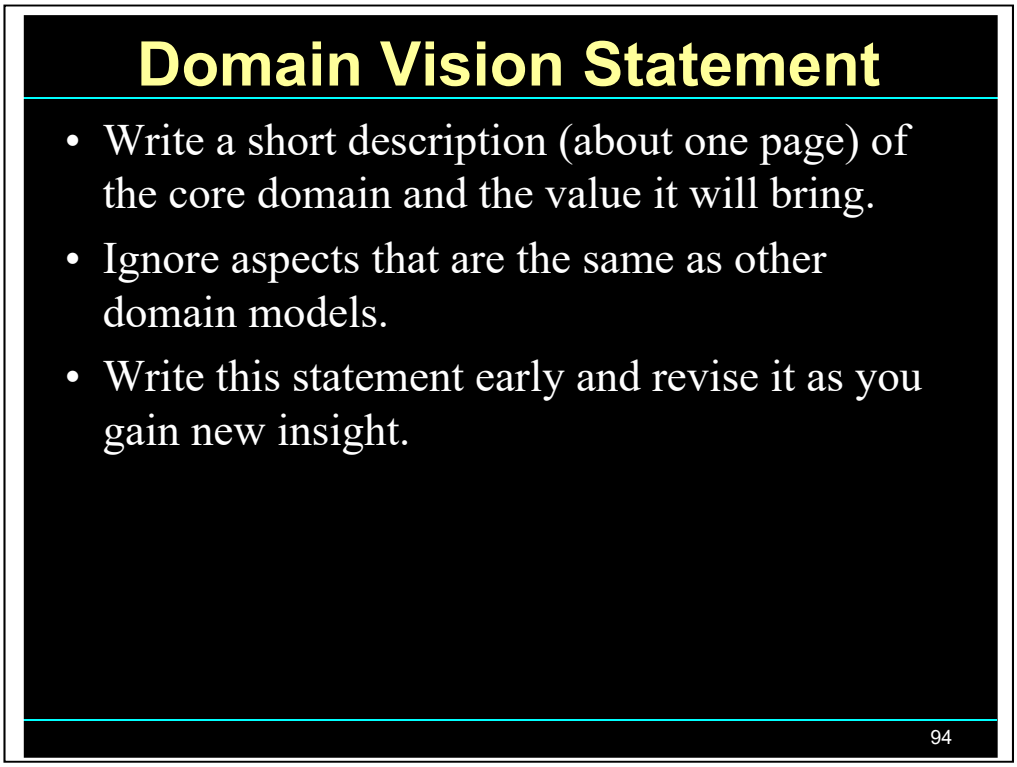
Generic Subdomain: Identify cohesive subdomains that are not the motivation for your project. Factor out generic models of these subdomains and place them in separate modules. These domains are not as important as the core domains. Don't assign your best developers to them. Consider off-the-shelf solutions or a published model. E.g. calendar, Invoicing, Accounting

Supporting Subdomain: Related to our domain but not worth spending millions of dollars every year. E.g. Advts on bank statements.

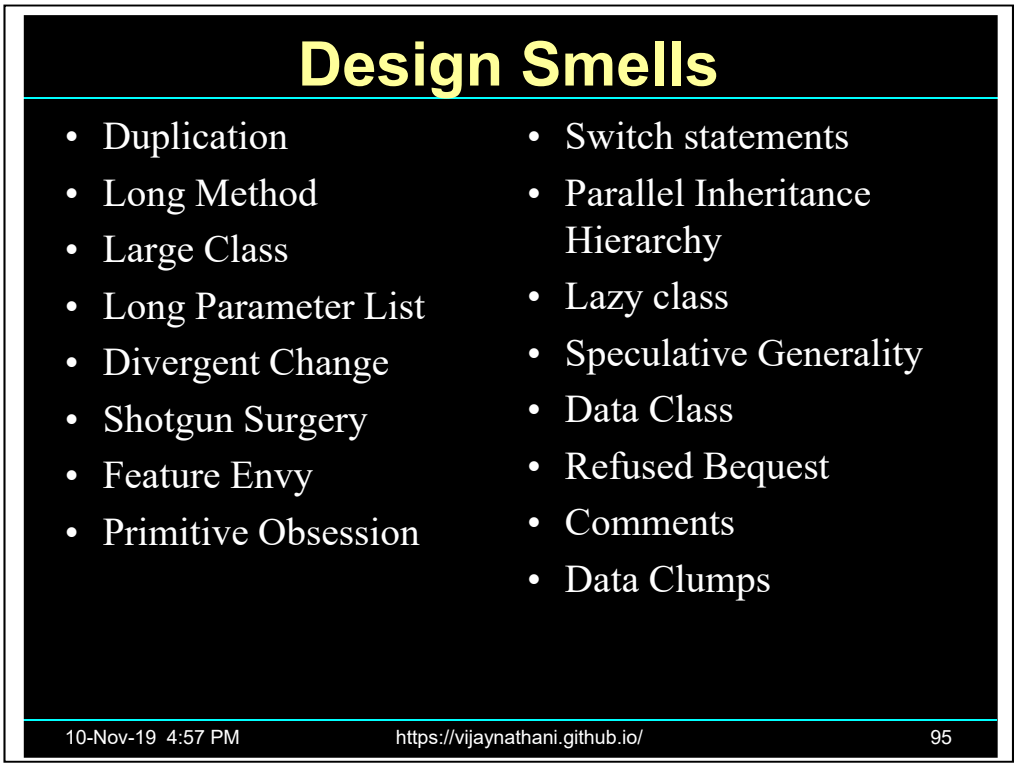
User ratings on Amazon and Ebay. On Amazon, this feature is supporting domain because customers usually buy the book even if others have given it low rating and they really want it. For ebay it is core domain because people will hesitate to make a deal with a person with low rating.



From Domain Driven Design.



Highlighted Core: Write a very brief document (3-7 pages) that describes the core domain and the primary interactions between core elements.



Divergent change occurs when one class is commonly changed in different ways for different reasons. If you look at a class and say, "Well, I will have to change these three methods every time I get a new database; I have to change these four methods every time there is a new financial instrument," you likely have a situation in which two objects are better than one. That way each object is changed only as a result of one kind of change.

Shotgun surgery is similar to divergent change but is the opposite. You whiff this when every time you make a kind of change, you have to make a lot of little changes to a lot of different classes. When the changes are all over the place, they are hard to find, and it's easy to miss an important change.

Feature Envy: The whole point of objects is that they are a technique to package data with the processes used on that data. A classic smell is a method that seems more interested in a class other than the one it actually is in.

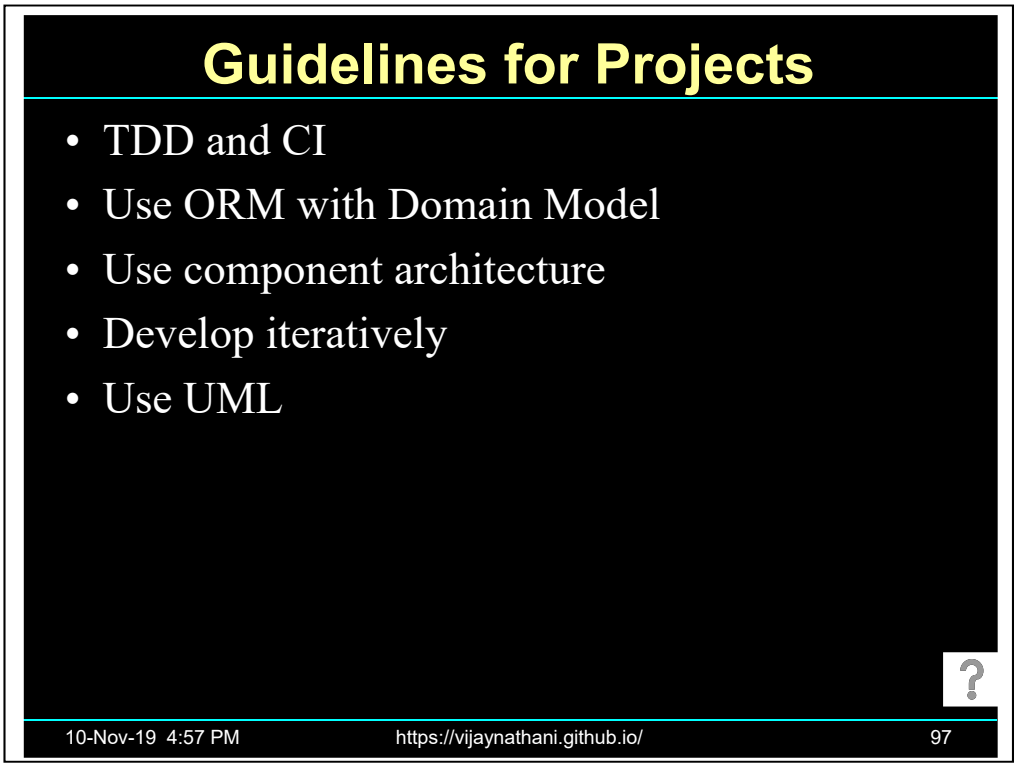
Data Clumps:Data items tend to be like children; they enjoy hanging around in groups together. Often you'll see the same three or four data items together in lots of places: fields in a couple of classes, parameters in many method signatures. Bunches of data that hang around together really ought to be made into their own object.

Data classes are like children. They are okay as a starting point, but to participate as a grownup object, they need to take some responsibility.

Refused Bequest: Subclasses get to inherit the methods and data of their parents. But what if they don't want or need what they are given? They are given all these great gifts and pick just a few to play with.

Project Design Smells

- Rigidity: The design is difficult to change.
- Fragility: The design is easy to break.
- Immobility: The design is difficult to reuse.
- Viscosity: It is difficult to do the right thing.
- Needless complexity: Over design.
- Needless repetition: Mouse abuse.
- Opacity: Disorganized expression.



QCardChips

QCoffee

Qinvestor

Q95 – Hangman (only in Java and PHP)

Q92 – Mastermind(only in Java and PHP)

Summary

- Keep it DRY, shy and tell the other guy.
- Prefer composition over inheritance
- Self-documenting code
- Not all parts of a large system will be well-designed.

No Golden Bullet.