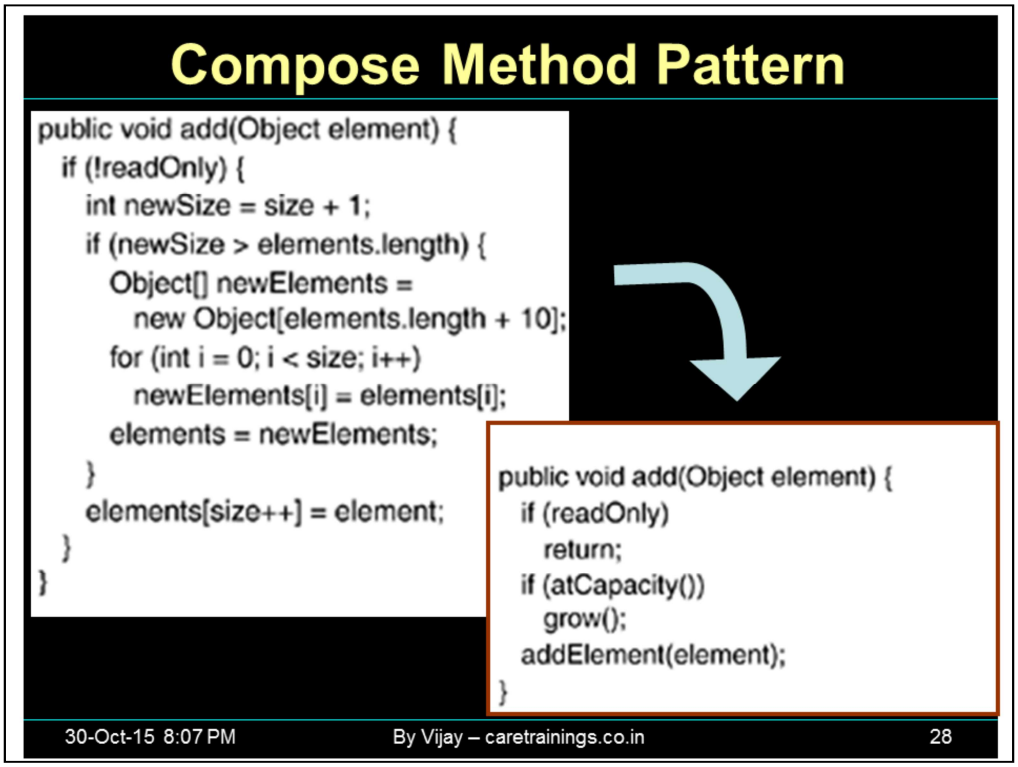
Code should readable

- Any fool can write code that a computer can understand. Good programmers write code that humans can understand. – Martin Fowler
  - ```
    java.util.Calendar c =
    java.util.Calendar.getInstance();
    c.set(2005,
     java.util.Calendar.NOVEMBER, 20);
    c.getTime(); OR
    ```
  - ```
    c = november(20, 2005) ;
    public void Date november (
         int day, int year)  { … }
    ```

Compare

```
void process() {
    input();
    count++;
    output();
}
```

```
void process() {
    input();
    tally();
    output();
}
private void tally(){
    count++;
}
```



Benefits and Liabilities

+Efficiently communicates what a method does and how it does what it does.

+Simplifies a method by breaking it up into well-named chunks of behavior at the same level of detail.

–Can lead to an overabundance of small methods.

–Can make debugging difficult because logic is spread out across many small methods.

Improve

```
flags |= LOADED_BIT;
```

- Solution: Extract to a message

```
void setLoadedFlag() {
    flags |= LOADED_BIT;
}
```

Improve Code

```
// Check to see if the employee
// is eligible for full benefits
if ((employee.flags & HOURLY_FLAG)
        && (employee.age > 65)) …


if (employee.
        isEligibleForFullBenefits())
```

Improve

```
public static void endMe() {
    if (status == DONE) {
        doSomething();
        …
        return
    } else {
        <other code>
    }
}
```

Avoid “else”

```
public static void endMe() {
    if (status == DONE) {
        doSomething();
        …
        return;
    }
    <other code>
}
```

Improve

```
public Node head() { …
    if (isAdvancing())
        return first;
    else
        return last;
}

  public static Node head() { …
      return isAdvancing()?first:last;
  }
```

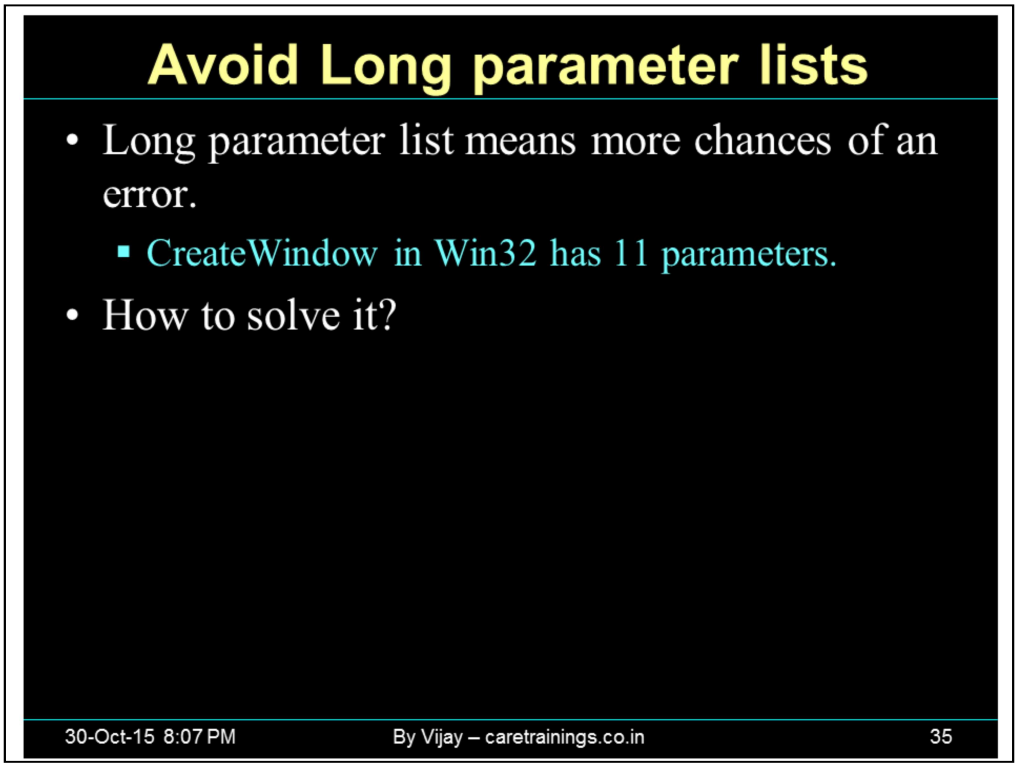
Improve

- Function signature

```
void render(boolean isSuite)
```

- Remove boolean variables from functions. Have two functions.

```
void renderForSuite()
void renderForSingleTest()
```

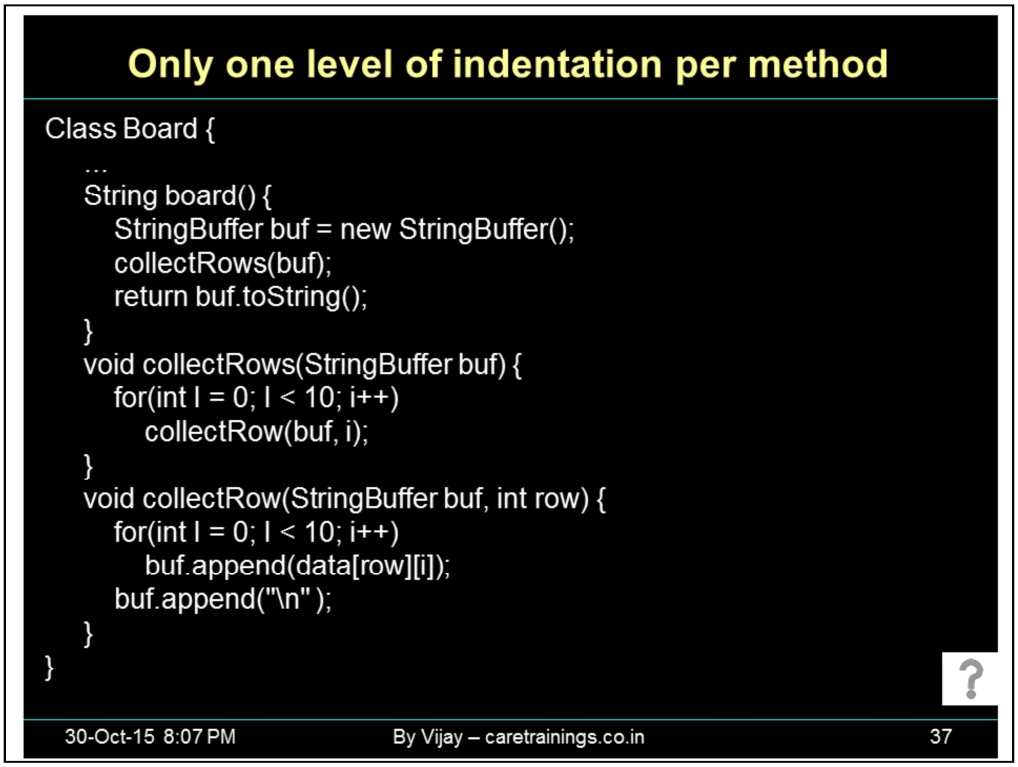


Break the method or

Create Helper class to hold parameters

Improve

```
class Board {
    ...
    String board() {
        StringBuffer buf = new StringBuffer();
        for(int i = 0; i < 10; i++) {
            for(int j = 0; j < 10; j++)
                buf.append(data[i][j]);
            buf.append("\n" );
        }
        return buf.toString();
    }
}
```



Self documenting code

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

Q22 – complexPassword; Q23 – TokenStream; Q25 - orderItems

Good Comments?

```
String text = "'''bold text'''";
ParentWidget parent = new BoldWidget(
          new MockWidgetRoot(), "'''bold text'''");
AtomicBoolean failFlag = new AtomicBoolean();
failFlag.set(false);
//This is our best attempt to get a race condition
//by creating large number of threads.
for (int i = 0; i < 25000; i++) {
    WidgetBuilderThread widgetBuilderThread = new
         WidgetBuilderThread(widgetBuilder, text,
              parent, failFlag);
    Thread thread = new Thread(widgetBuilderThread);
    thread.start();
}
assertEquals(false, failFlag.get());
```

30-Oct-15 8:07 PM By Vijay – caretrainings.co.in 38

Comment to

• explain WHY we are doing something?

•Also for external documentation. Javadocs public API

•To give warnings: e.g. Don't run unless you want to kill this program.

•Todo comments

Find the flaw

```
if (deletePage(page) == E_OK)
    if (registry.deleteReference(page.name) == E_OK)
       if ( configKeys.deleteKey(
              page.name.makeKey()) == E_OK)
            logger.log("page deleted");
       else
            logger.log("configKey not deleted");
    else
       logger.log ("deleteReference … failed");
```

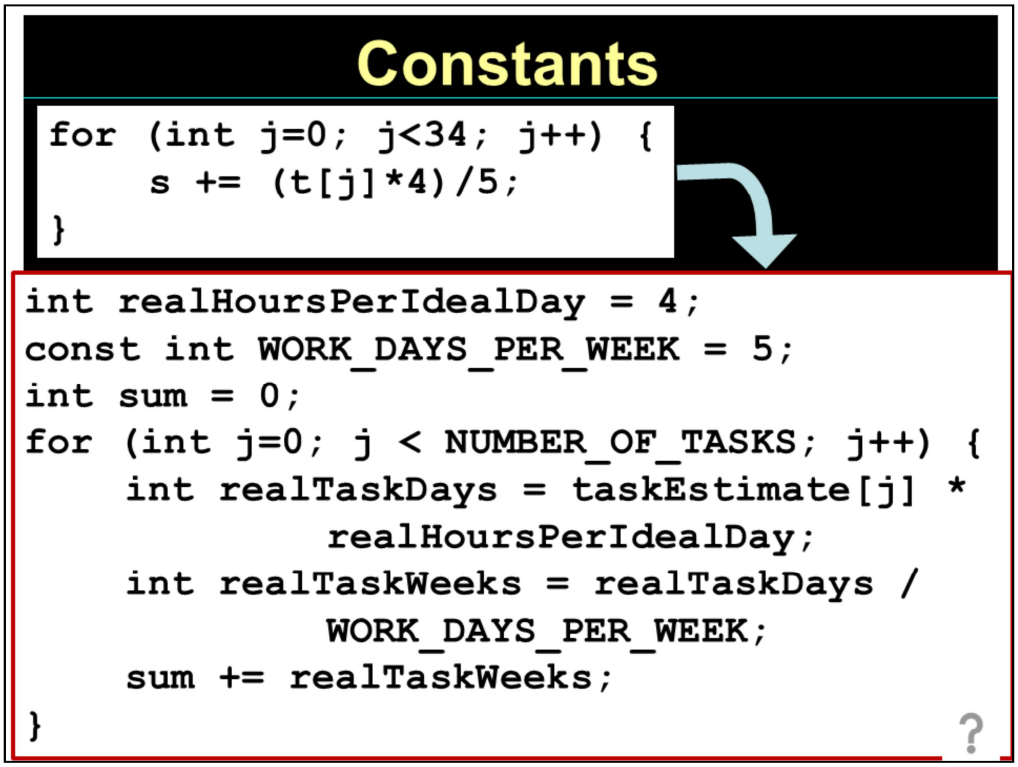
```
try {
    deletePage(page);
    registry.deleteReference(page.name);
    configKeys.deleteKey(
                   page.name.makeKey());
} catch (Exception e) {
    logger.log(e.getMessage()); }
```

Samurai Principle

- Throw exception if any error occurs.

Guidelines

- Prefer long to int and double to float to reduce errors.
- Avoid literal constants other than “”, null, 0 and 1.



Q32 – FoodSalesReport.

Fail-Fast

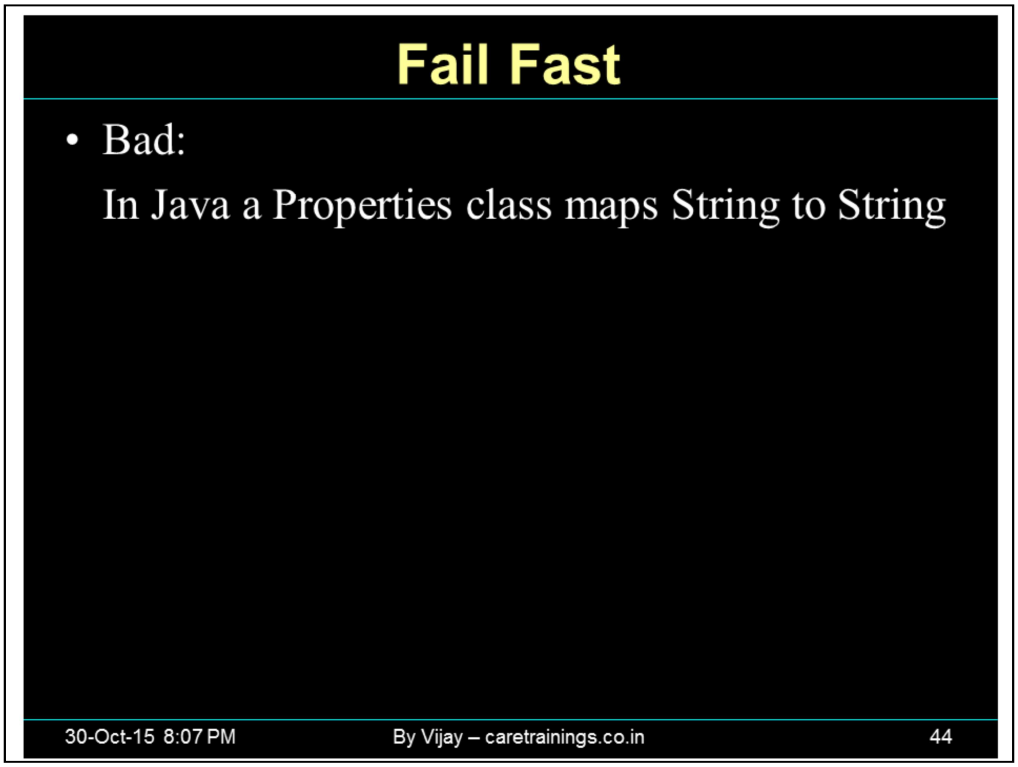
- Compile time checking is best

  Contrast this signature:

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

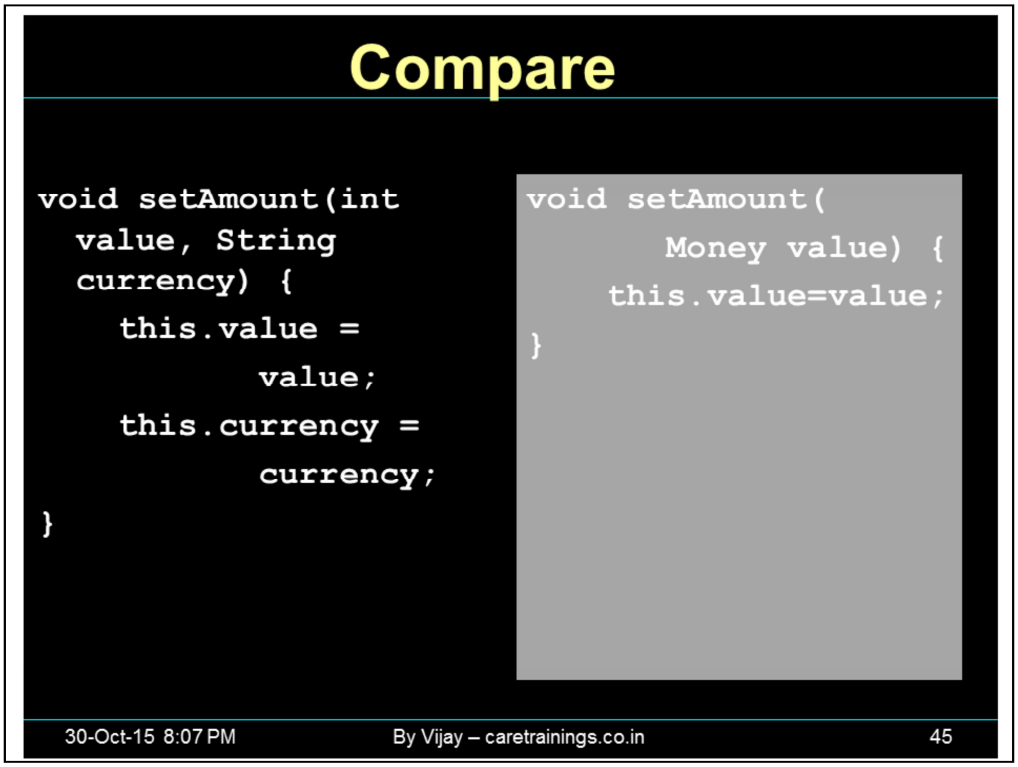
  with

```
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

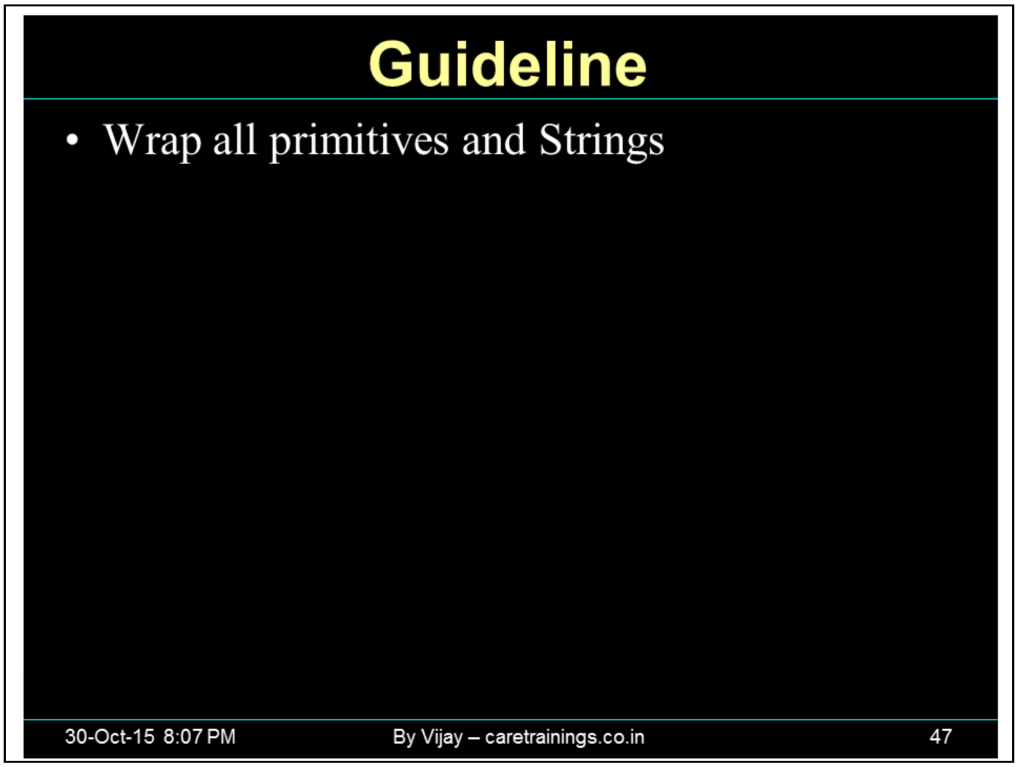
•Less prone to error, because the caller does not have to worry about exceptions

Improve

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

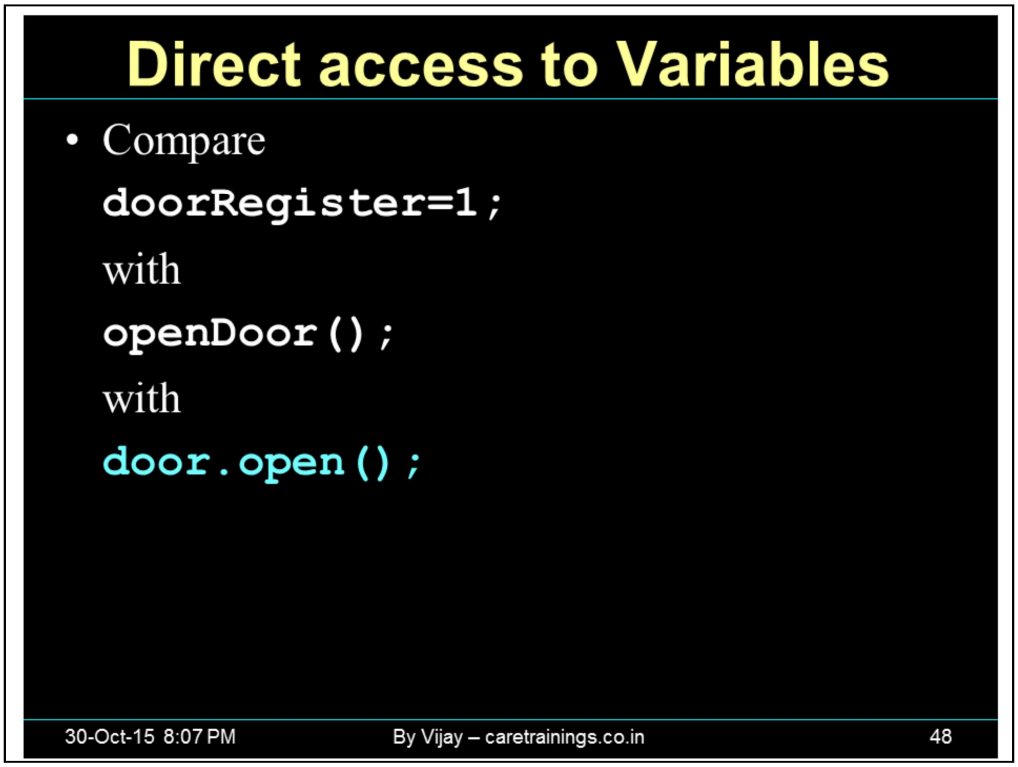
- Solution: Use Parameter Object

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



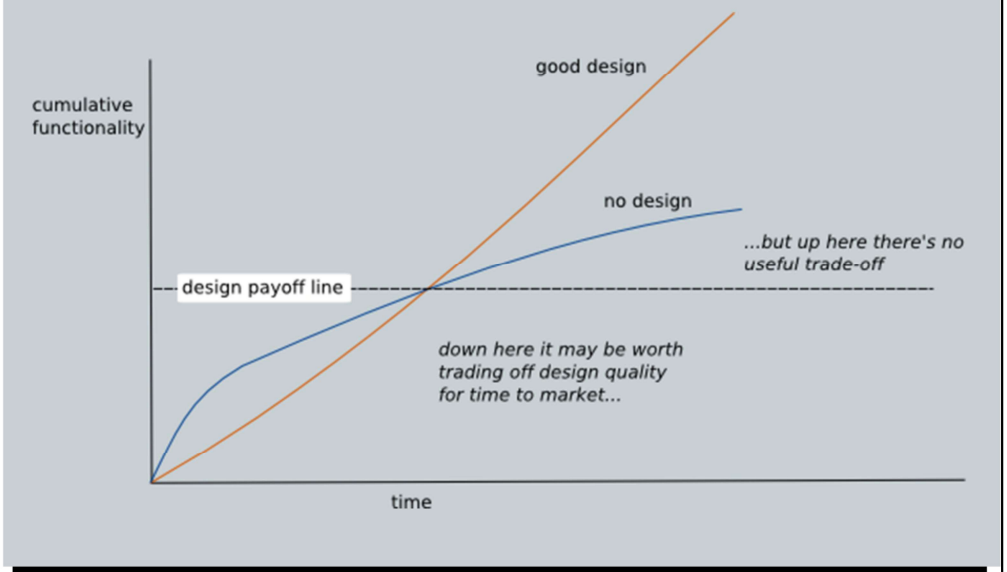
An int on its own is just a scalar with no meaning. When a method takes an int as a parameter, the method name needs to do all the work of expressing the intent. If the same method takes an hour as a parameter, it's much easier to see what's happening. Small objects like this can make programs more maintainable, 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.

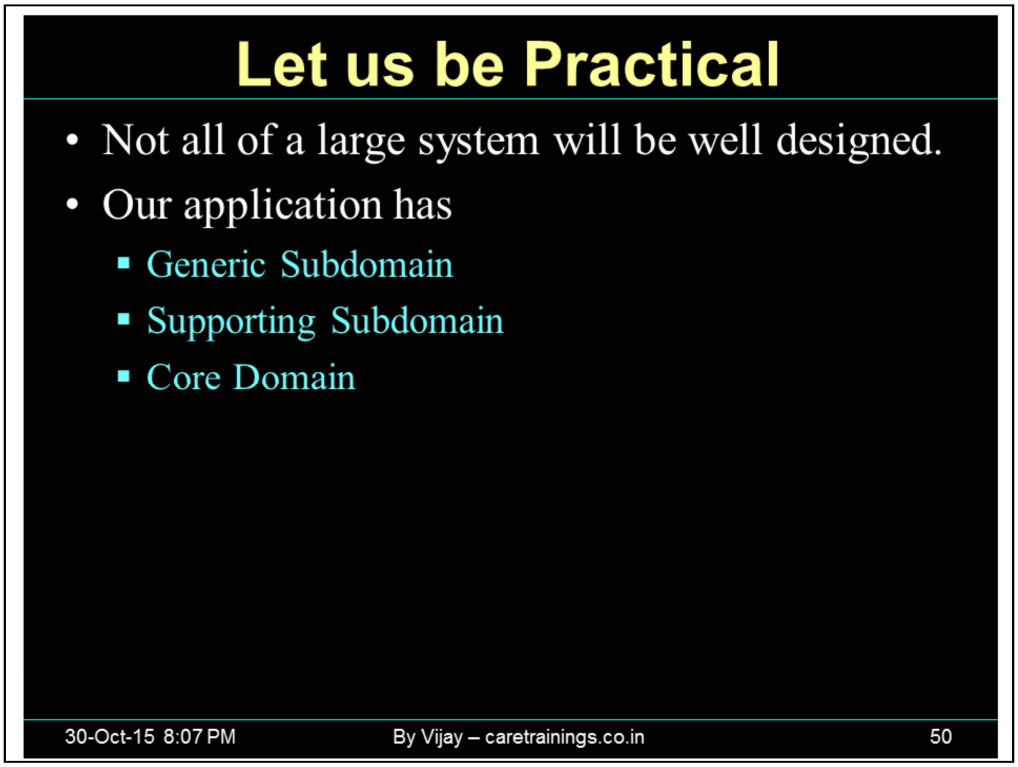
Small objects such as hour or money also give you an obvious place to put behavior that otherwise would have been littered around other classes. This becomes especially true when you apply the rule relating to getters and setters and only the small object can access the value.



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

Is Good Design worth it?





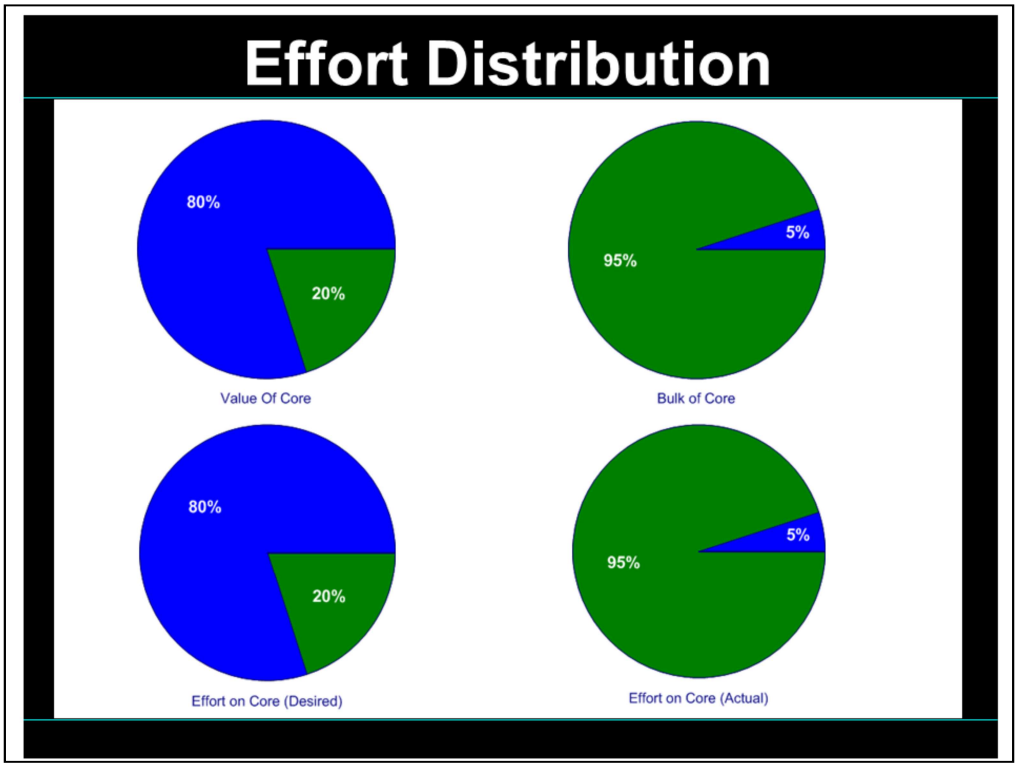
Core domain:

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.

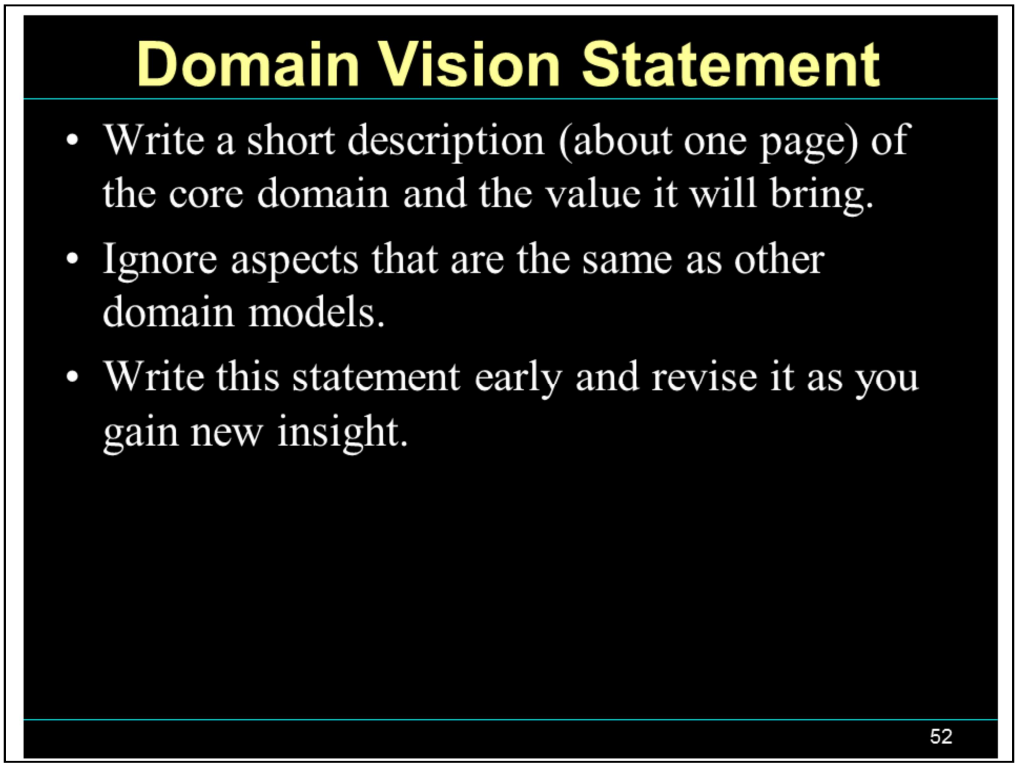
Generic Subdomain: Identify cohesive subdomains that are not the motivation for your project. Factor out generic models of these subdomains and place them in separate modules. These domains are not as important as the core domains. Don't assign your best developers to them. Consider off-the-shelf solutions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

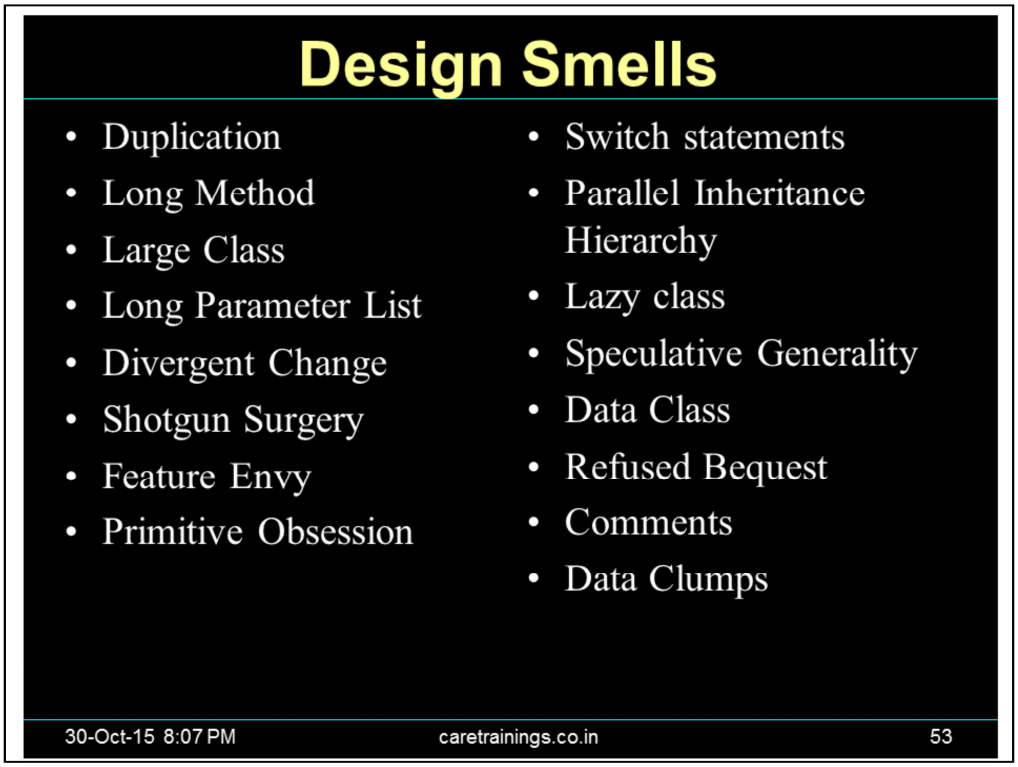
User ratings on Amazon and Ebay. On Amazon, this feature is supporting domain because customers usually buy the book even if others have given it low rating and they really want it. For ebay it is core domain because people will hesitate to make a deal with a person with low rating.



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.

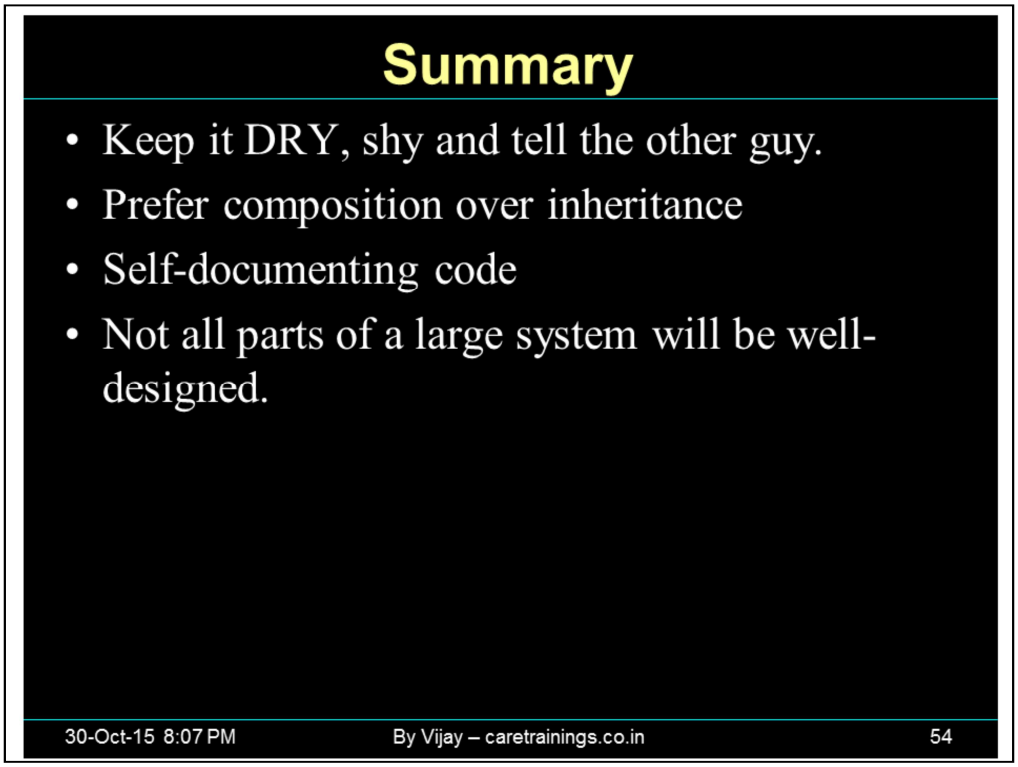
Shotgun surgery is similar to divergent change but is the opposite. You whiff this when every time you make a kind of change, you have to make a lot of little changes to a lot of different classes. When the changes are all over the place, they are hard to find, and it's easy to miss an important change.

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.



No Golden Bullet.