

Package by feature

Kostis Kapelonis (kkapelon@gmail.com)

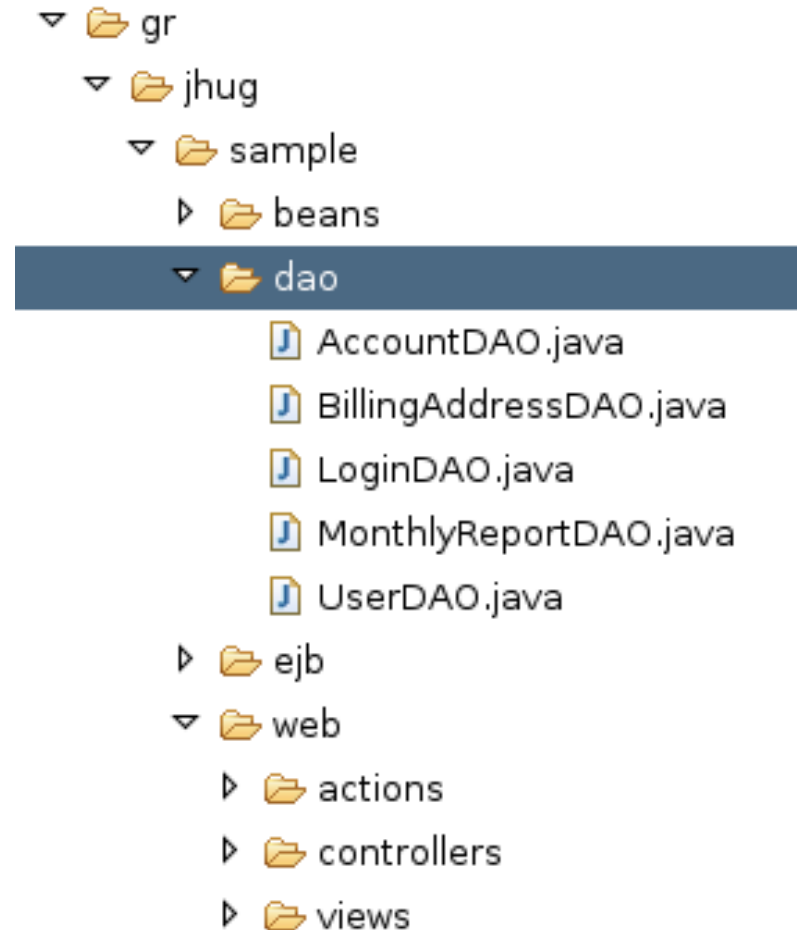


My constant Java complaint



Java packages by Layer

- ▶ Dao, GUI, WEB
- ▶ Model, View, Controller
- ▶ Client, Server, Database
- ▶ Backend, Front-end
- ▶ Service (!!!)
- ▶ Managers, Actions, Flows e.t.c

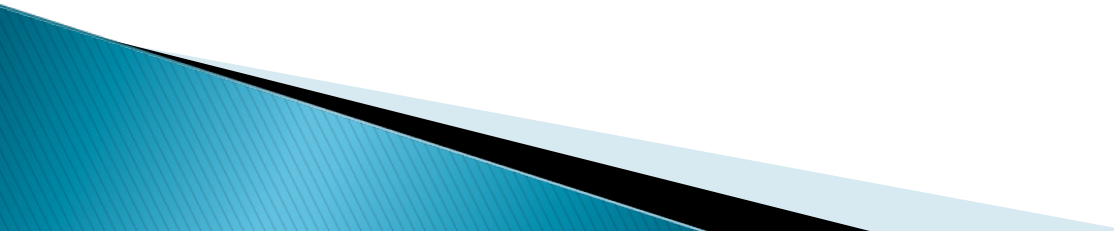


You are doing it wrong!

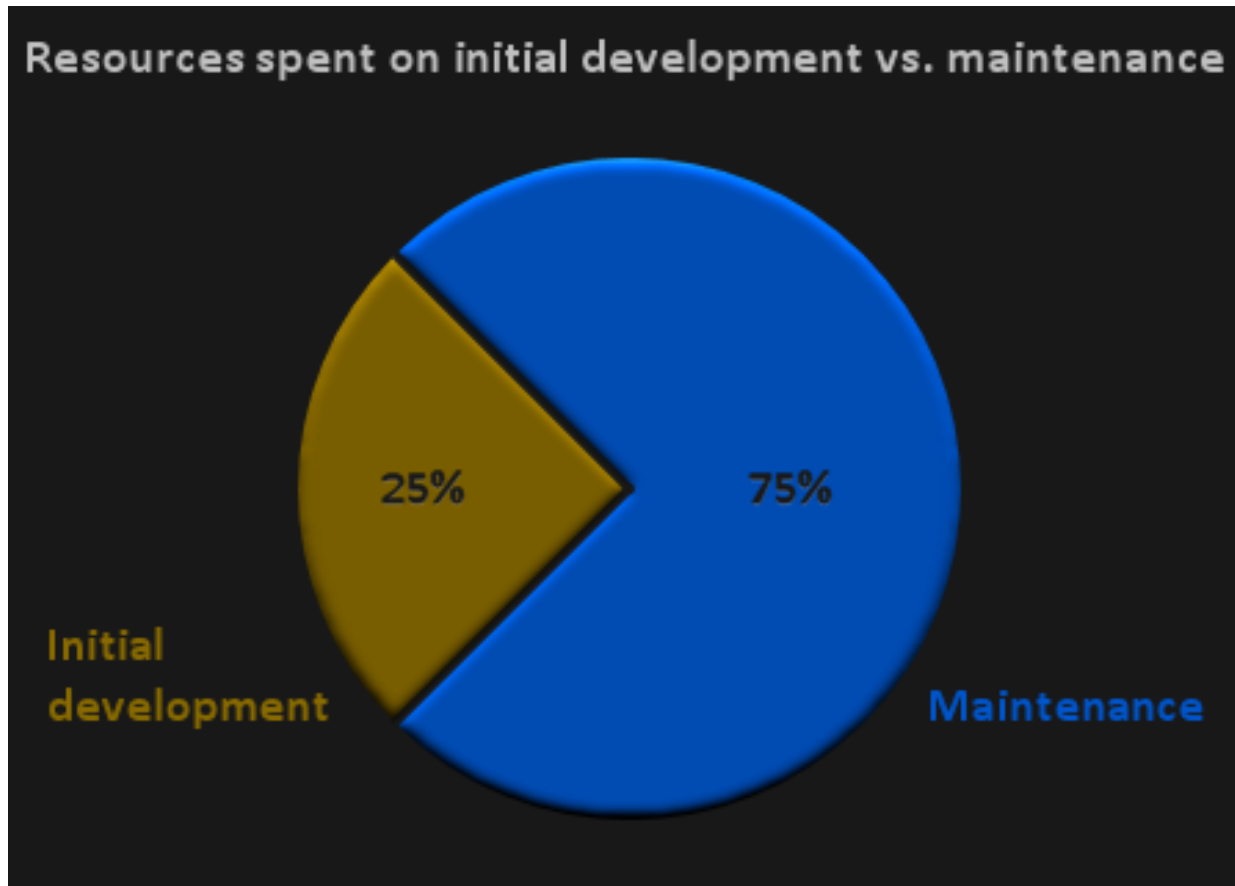


(At least in my humble opinion!)

We are used to layers

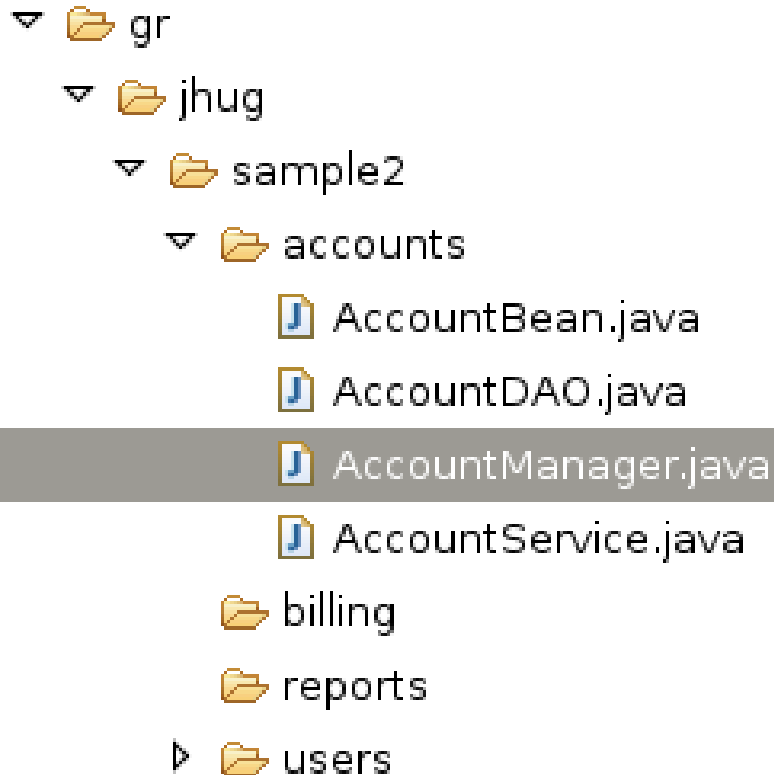
- ▶ Programmers see layers everywhere
 - ▶ We like to think of abstractions
 - ▶ We *might* change the GUI layer (do we?)
 - ▶ We *might* change the ORM (do we?)
 - ▶ It helps during initial stages
 - ▶ I admit personally that I have coded like this
 - ▶ Most straightforward way to code
 - ▶ It is not helpful during maintenance
- 

Think the future today!



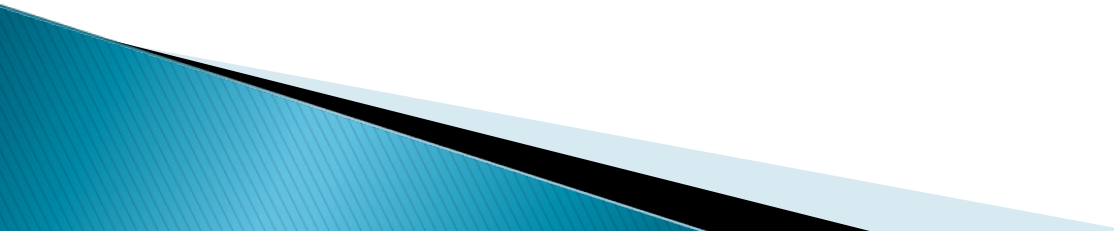
- ▶ Numbers vary from 60% to 80%

Package by feature



- ▶ Think in business terms
- ▶ Assume you are a client
- ▶ Accounts, billing, security, budget, reports, users, documents,
- ▶ Everything inside!

Think “maintenance”

1. Clients always request features (not layers)
 2. Project grows horizontally (feature scope)
 3. Encapsulation (good OOP)
 4. Plugin system/Lego like development
 5. Good architecture (Quality wise)
- 

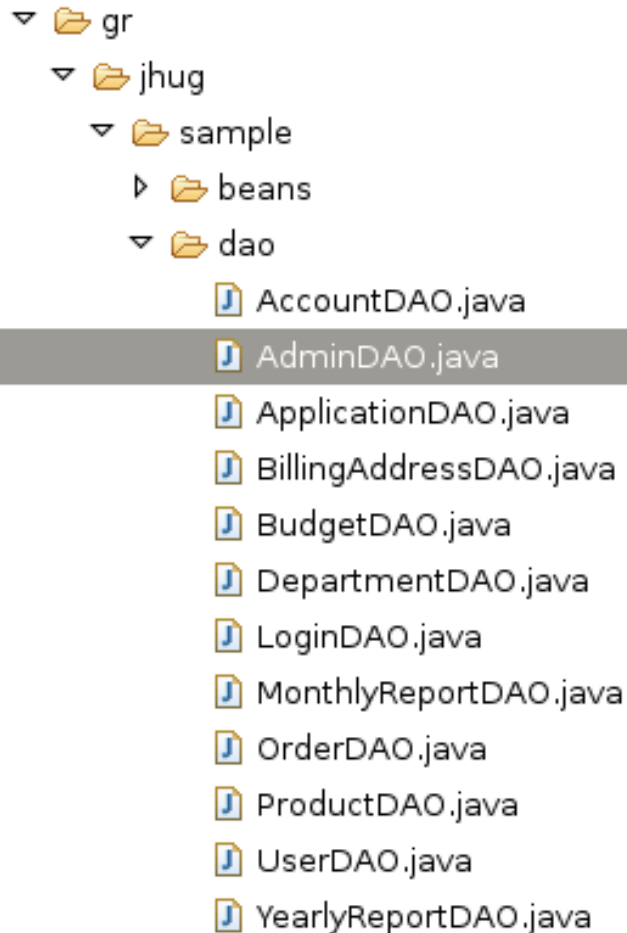
1. Clients think in features



The usual scenario

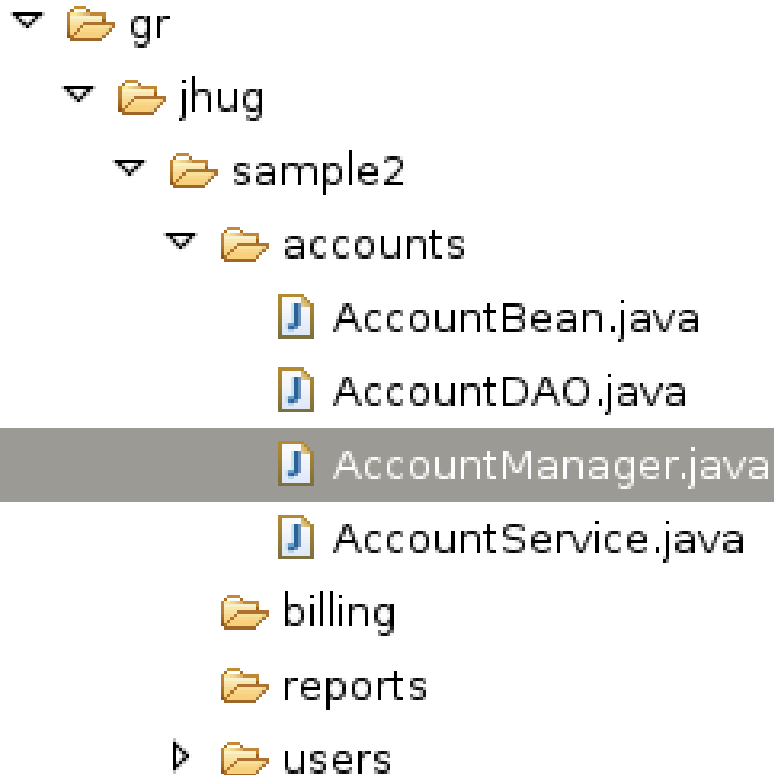
- ▶ You are a senior Java developer
- ▶ You go to a NEW company/project
- ▶ You are assigned your first issue:
 - “There is a bug in the billing calculation”
 - “This report is wrong”
 - “The budget screen is missing a button”
 - “The document is not saved”
 - “Change the colour in the login screen”

Is this easy?



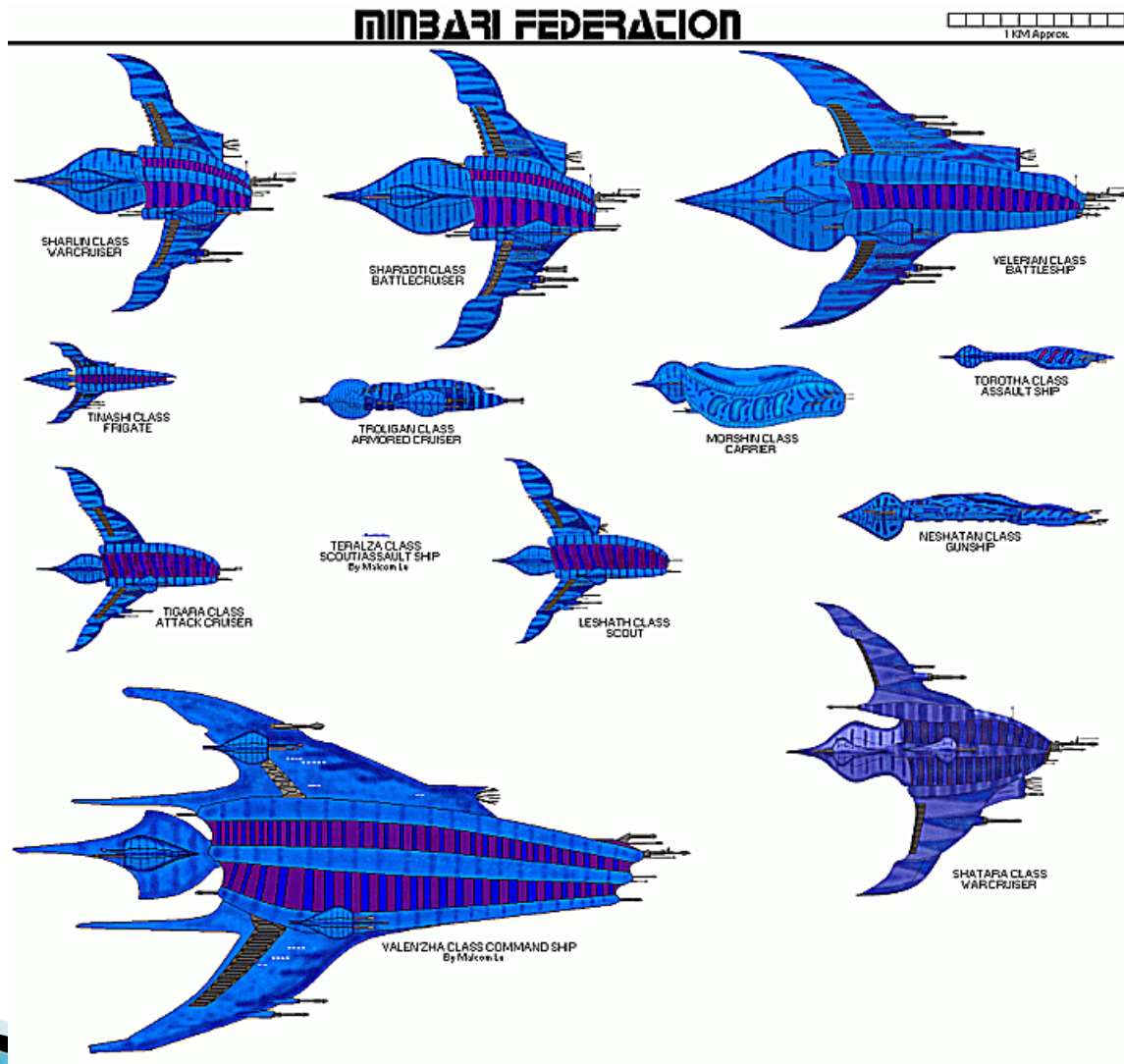
- ▶ You open Eclipse to see the code
- ▶ NOTHING makes any sense to you
- ▶ Where do you start?
- ▶ Usually you ask another developer

A better alternative

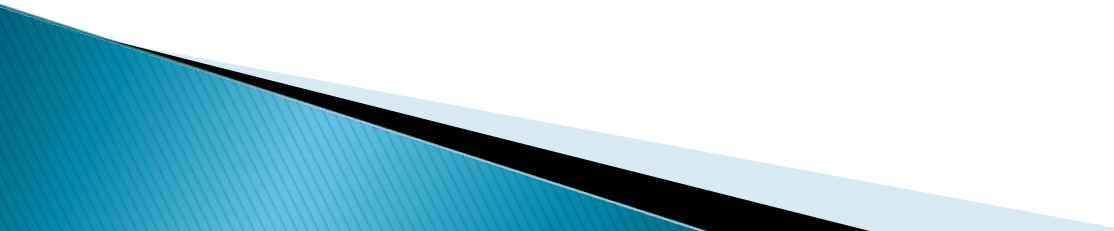


- ▶ Instant detection of affected code
- ▶ Changes contained in the package
- ▶ No need to look at the rest of the code
- ▶ Great for isolating junior developers

2. Project size












Growing project code size

- ▶ Assume you have two enterprise projects
 - ▶ The second could be just a newer version
 - ▶ First project is 100.000 lines of code
 - ▶ Second project is 1.000.000 lines of code
 - ▶ How do they look in Eclipse?
- 

Package by layer

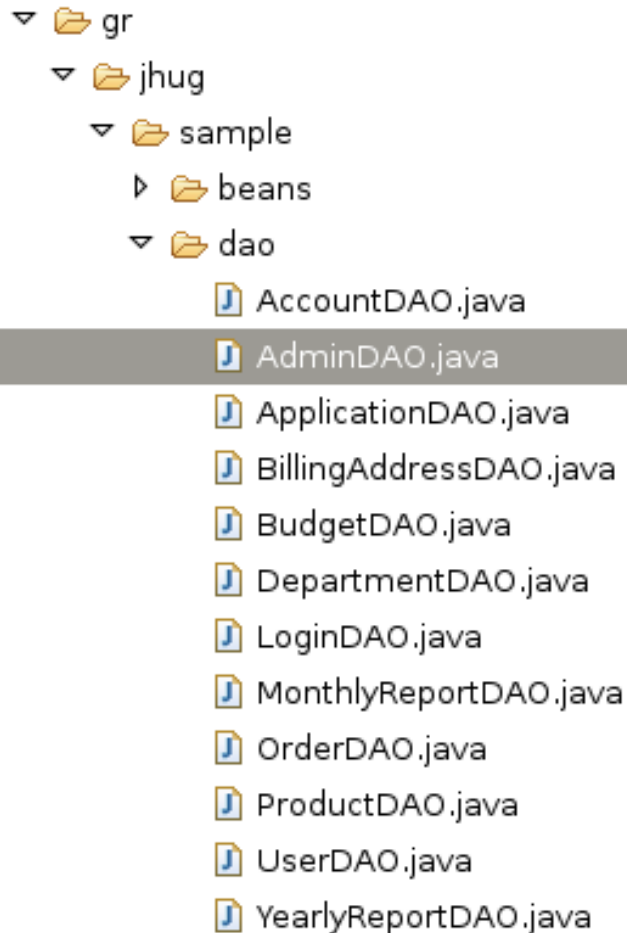
Project 1

- ▼  gr
 - ▼  jhug
 - ▼  sample
 - ▶  beans
 - ▶  dao
 - ▶  ejb
 - ▼  web
 - ▶  actions
 - ▶  controllers
 - ▶  views

Project 2

- ▼  gr
 - ▼  jhug
 - ▼  sample
 - ▶  beans
 - ▶  dao
 - ▶  ejb
 - ▼  web
 - ▶  actions
 - ▶  controllers
 - ▶  views








Package by layer
















- ▶ Thousands of “actions”, DAOs
- ▶ Usually alphabetically sorted
- ▶ Very hard to work with.
- ▶ Cause for code duplication

Package by feature

Project 1

- ▼  gr
 - ▼  jhug
 - ▼  sample2
 - ▶  accounts
 -  billing
 -  orders
 - ▶  users

Project 2

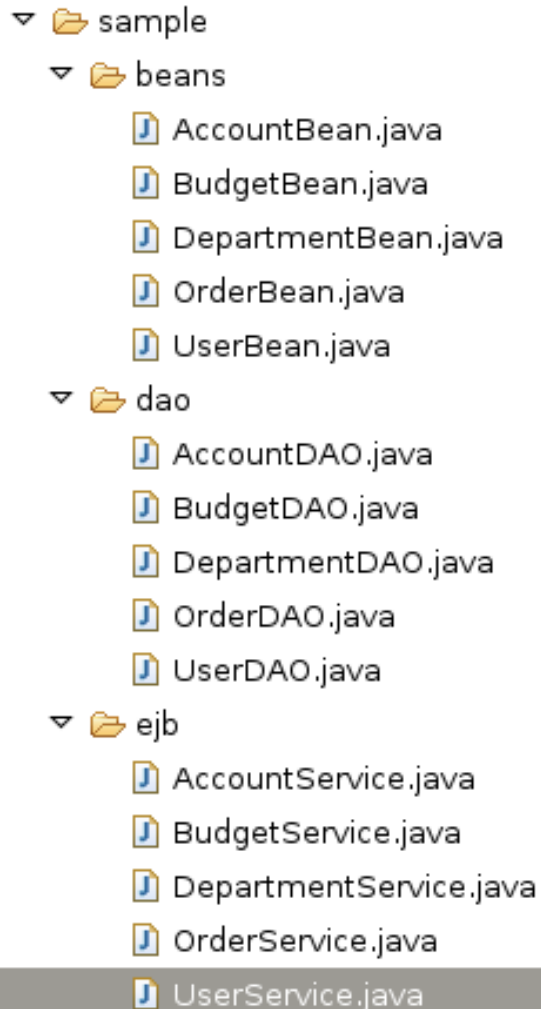
- ▼  gr
 - ▼  jhug
 - ▼  sample2
 - ▶  accounts
 -  admin
 -  billing
 -  categories
 -  inventory
 -  merchants
 -  orders
 -  products
 -  security
 - ▶  users

3. Class encapsulation



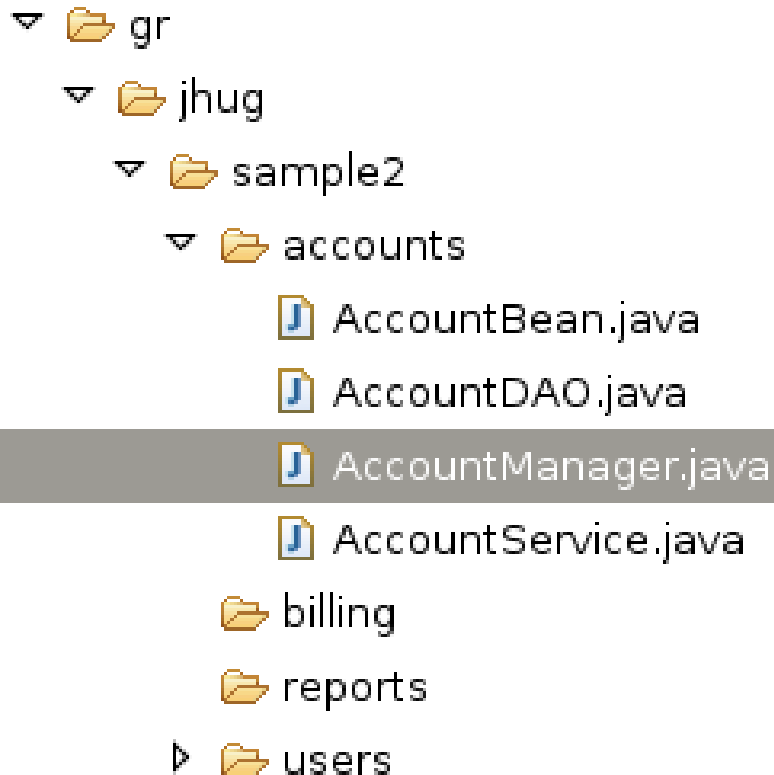
- ▶ OOP in package level

All classes are public



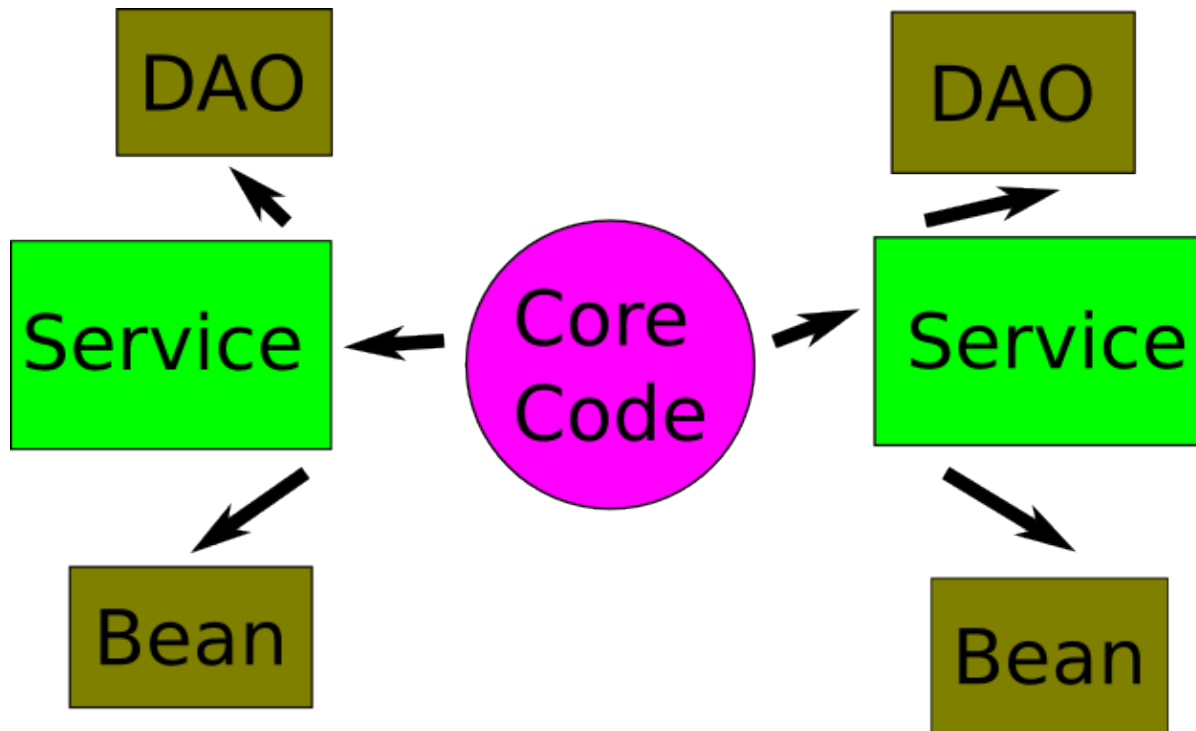
- ▶ Service uses DAO
- ▶ Service uses Bean
- ▶ DAOs are public!
- ▶ Junior programmers could go directly to DAO instead of using the service

Package private visibility

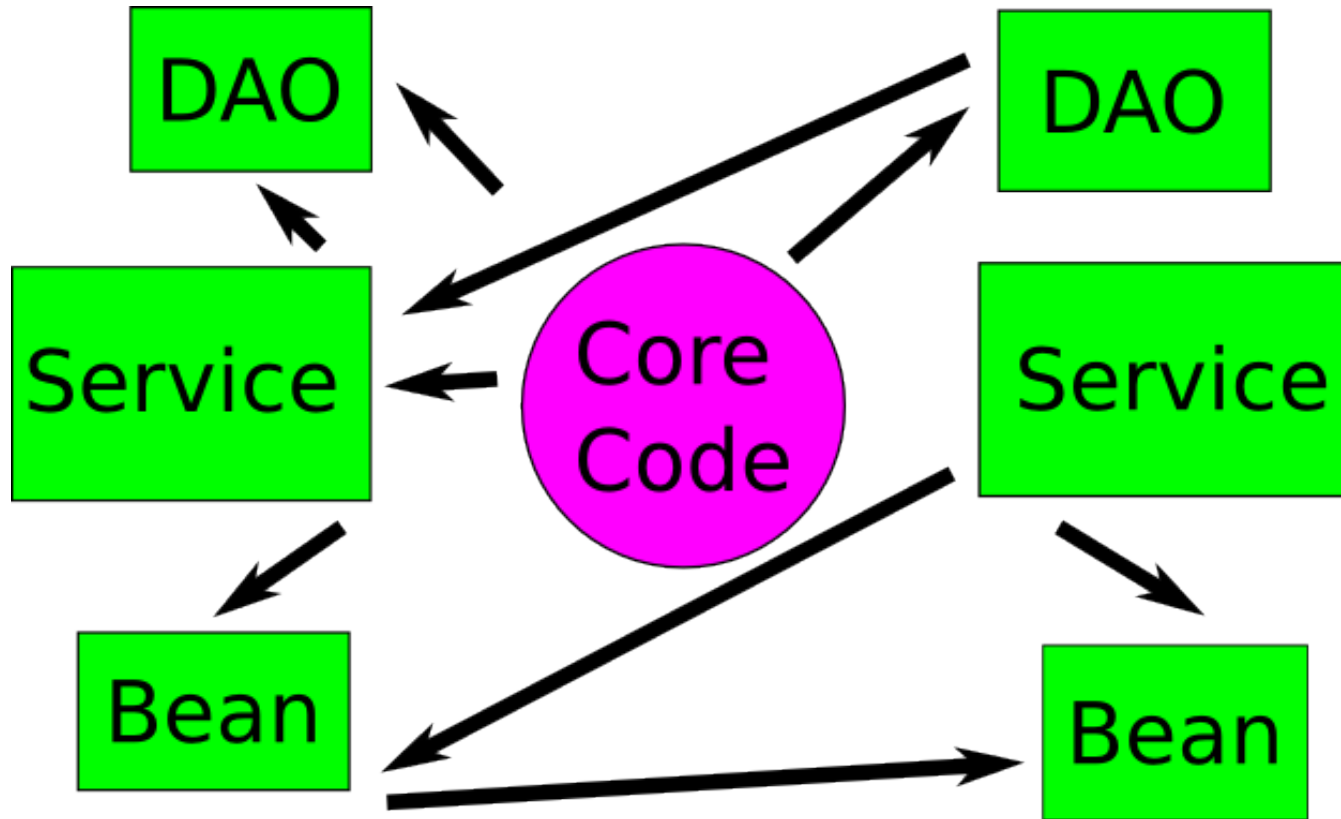


- ▶ DAOs are package private
- ▶ Beans *could* be package private as well.
- ▶ Everybody is forced to use the Services

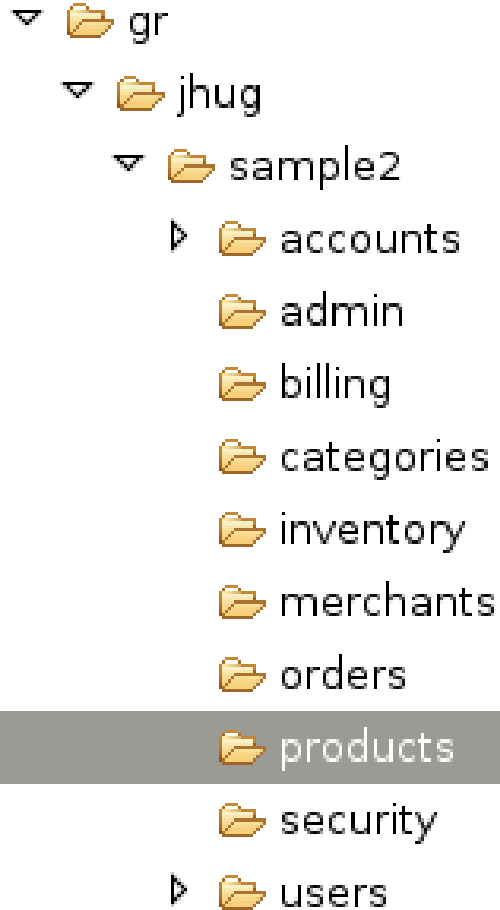
Good architecture



Bad architecture

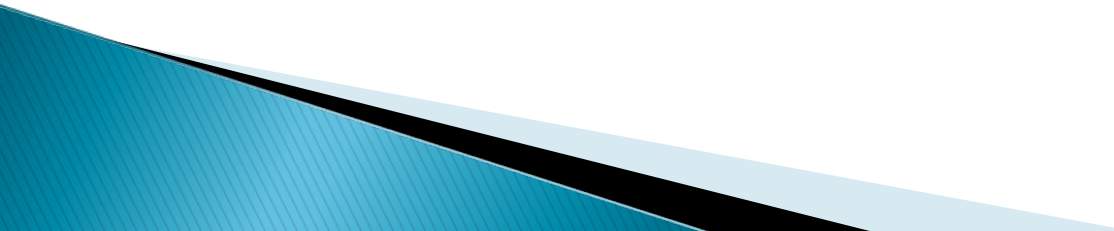


Reuse code easily



- ▶ Packages are self-contained!
- ▶ They can be added in projects
- ▶ They can be removed
- ▶ They can be converted to jars/wars/ears/OSGI e.t.c

5. Avoiding package cycles

- ▶ A cycle happens when package A uses package B and package B uses package A
 - ▶ Also transitive $A \rightarrow B \rightarrow C \rightarrow A$
 - ▶ Package cycles are BAD!
 - ▶ They make refactoring difficult.
 - ▶ Once must change ALL packages at ONCE.
 - ▶ Detected by common quality tools Jdepend, Sonar, CAP e.t.c.
- 

Real world example

Cycles

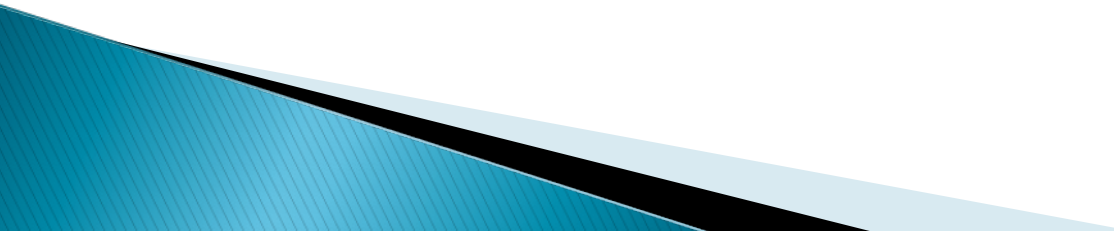
[[summary](#)] [[packages](#)] [[cycles](#)] [[explanations](#)]

Package	Package Dependencies
org.dbunit	org.dbunit.operation org.dbunit.dataset.datatype org.dbunit.dataset org.dbunit.dataset.datatype
org.dbunit.ant	org.dbunit.util org.dbunit.database org.dbunit.util
org.dbunit.assertion	org.dbunit.dataset.datatype org.dbunit.dataset org.dbunit.dataset.datatype
org.dbunit.database	org.dbunit.util org.dbunit.database
org.dbunit.database.search	org.dbunit.util org.dbunit.database org.dbunit.util
org.dbunit.database.statement	org.dbunit.database org.dbunit.util org.dbunit.database

My second complaint



Design pattern overload

- ▶ Normally a programmer should face a SPECIFIC problem, consult the GOF book and THEN apply the pattern.
 - ▶ In reality programmers look at the book and find cool design patterns they want to add in their CV.
 - ▶ Singletons are evil (look it up)
 - ▶ Also factories are obsolete (see dependency injection)
- 

Why this is bad

- ▼  web
 -  controllers
 -  delegates
- ▼  facade
 -  builders
 -  factories
- ▼  proxies
-  singletons

- ▶ Similar as before
- ▶ Delegates, factories, builders, proxies.
- ▶ These have NO meaning for the actual code.

An alternative approach

▼ accounts

- AccountBean.java
- AccountDAO.java
- AccountDelegate.java
- AccountFactory.java
- AccountManager.java
- AccountProxy.java
- AccountService.java

- ▶ Just append the pattern to the class name.
- ▶ Still name the packages by layer

The End

