Technical Programming II Handbook

Boniface Kabaso, Xolisa Piyose

Published with GitBook

Table of Contents

Introduction

Welcome to 3rd year National Diploma in Information Technology. Technical Programming II/III is one of four the subjects offered in your 3rd year and is an elective taken by students of the Software Development Domain. This course is technology centric, designed to teach you essential enterprise appli- cation development skills, coupling the most current programming techniques with sound industry practices.

This course will equip you with skills to handle web, mobile and desktop application planning, design, and implementation of a technical architecture using various open source frameworks and technologies that are used in the industry currently.

Java Programming language will be used as a programming vehicle to de- liver this course. Refresher Java tutorial notes will be provided on WEBCT. Please note that this course could also be delivered using C#, Python, Ruby or even PHP.

The course is being delivered in Java because in second year you picked up Java skills.

Study Units and Themes

This course is technology centric, designed to teach students essential enter- prise application development skills, coupling the most current programming techniques with sound industry practices.

This course will equip students with skills to handle web, mobile and desktop application planning, design, and implementation of a technical architecture using various open source frameworks and technologies that are used in the industry currently.

Chapter No	Theme/Topic	Specific Outcome
First Semester		
First Term (I)		
1	Software Development Infrastructure (Maven and Git)	Use Build tools and Version control
2	Testing, Test Driven Development	Use software Industry best practice Test driven approach to increase flexibility and testability of applications you build
3	Java Data Structures Concepts Recap (Generic Collections)	Recap of the Java Interfaces, Concrete and Abstract classes, and Collections Framework
4	Application Design Concepts and Principles	Learn and apply key principles that facilitate repeatable, quality designs such as the Liskov Substitution Principle, the Law of Demeter,High Cohesion, Loose Coupling, and many others
5	Design Patterns and Refactoring	Use Design Patterns to design and develop flexible and scalable applications that exchange messages with other systems
Second Term (II)		
6	Domain Driven Design	Know the central premise of Domain-Driven Design

7	Microservice Architectures	Learn Middle Tier Technologies and largely REST based Web Services
8	Application Security	Secure Enterprise applications by URL and role, and integrate other authentication APIs.
Second Semester		
Third Term (III)		
9	Front end Mobile Platform Development	Building Android GUIs and related components
Fourth Term (IV)		
10	Communication Services Mobile Platform Development	Design and implement applications that can be deployed on cellphones and communicate with enterprise backend systems

License

Chapter 1: Software Development Infrastructure

Chapter Objectives

1. Demostrate the ability to set up a development infratructure
2. Understand the Source code version Systems available to software Development process
3. Understand IDEs and build systems used in the development of Software.

Development Infrastructure

The productivity of the software industry is defined by an infrastructure that helps the developers to produce quality software from the word go. The infrastructure has a lot of faces to it, ranging from a simple IDE to complex platform like deployment cloud environment. This section introduces you to things you need to put in place if you want to develop quality software in a collaborative way or if you need to open source your personal project for the rest of the world community to help you code it much faster and provide the feedback and the quality required to be built into any software product.

Integragated Development Platform (IDEs)

Integrated Development Environments (IDE) provide benefits to programmers that plain text editors cannot match. IDEs can parse source code as it is typed, giving it a syntactic understanding of the code, help you with establishing coding standards and many more feature. This allows advanced features like code generators, auto-completion, refactoring, and debuggers. This course is going to use Netbeans for development, but note that there are other IDEs that can be used for executing the same task

IntelliJ IDEA

IntelliJ IDEA is a commercial IDE with a big following that swear by it. It has excellent Java support. It isextensible via plugins. Its standout feature is the outstanding refactoring support. It provides a robust combination of enhanced devel- opment tools, including: refactoring, J2EE support, Maven, Git, Ant, JUnit, and CVS integration. Packaged with an intelligent Java editor, coding assistance and advanced code automation tools, IDEA enables Java programmers to boost their productivity while reducing routine time consuming tasks.

Netbeabs IDE

Netbeans is a free IDE backed by Oracle Corporation. Netbeans is built on a plugin architecture, and it has respectable third-party vendor support . The main advantage of Netbeans its excellent GUI designer. It includes syntax highlighting and language support for Java, JSP, XML/XHTML, visual design tools, code generators, ant, Maven and CVS, Git, Mercurila and Subversion support.

Eclipse IDE

Eclipse is a free IDE that has taken the Java industry by storm. Built on a plugin architecture, Eclipse is highly extensible and customizable. Third-party vendors have embraced Eclipse and are increasingly providing Eclipse integration. Eclipse is built on its own SWT GUI library. Eclipse excels at refactoring, J2EE support, and plugin support. The only current weakness of Eclipse is its lack of a Swing or SWT GUI designer. The eclipse platform provides tool developers with ultimate flexibility and control over their software technology.

Others

There are tons of IDEs on the market today. Some notable ones include SublimeText, Visual Studio, Eclipse STS

What this ourse will use

While you can use any IDE of your choice in this course, most of the demonstrations and sample source code will be

delivered using JettBrainss IntelliJ IDEA. The is a student licence that you can obtain for free valid for on year. InttelJ IDEA Classroom License Requests

Source Code Management and Version Control

Version Control Basic Concepts

We begin with a discussion of general version control concepts, work our way into the specific ideas behind two major version control systems called Subversion and Git , and show some simple examples of Git in use. Even though the examples here show people sharing program source code, keep in mind that version control can manage any sort of file collection. It is not limited to helping computer programmers.

What is Version Control System

A version control system is a software package that when initiated will monitor your files for changes and allow you to mark the changes at different levels so that you can revisit those marked stages whenever needed.

When installed and initiated, a version control system creates a local directory at the same place where your files reside, which it uses to manage the entire history of the changes made to your files.

So why is this interesting? So far, this sounds like the definition of a typical file server. What makes the Subversion repository special is that it remembers every change ever written to it: every change to every file, and even changes to the directory tree itself, such as the addition, deletion, and rearrangement of files and directories. When a client reads data from the repository, it normally sees only the latest version of the filesystem tree. But the client also has the ability to view previous states of the filesystem. For example, a client can ask historical questions like,

1. What did this directory contain last Wednesday?
2. Who was the last person to change this file, and what changes did they make?

These are the sorts of questions that are at the heart of any version control system: systems that are designed to record and track changes to data over time.

The core mission of a version control system is to enable collaborative editing and sharing of data. But different systems use different strategies to achieve this.

The Problem of File-Sharing

All version control systems have to solve the same fundamental problem: how will the system allow users to share information, but prevent them from accidentally stepping on each other's feet? It's all too easy for users to accidentally overwrite each other's changes in the repository.

Consider this scenario. Suppose we have two co-workers, Harry and Sally. They each decide to edit the same repository file at the same time. If Harry saves his changes to the repository first, then it's possible that (a few moments later) Sally could accidentally overwrite them with her own new version of the file. While Harry's version of the file won't be lost forever (because the system remembers every change), any changes Harry made won't be present in Sally's newer version of the file, because she never saw Harry's changes to begin with. Harry's work is still effectively lost or at least missing from the latest version of the file and probably by accident.

This is definitely a situation we want to avoid!

The Lock-Modify-Unlock Solution

Many version control systems use a lock-modify-unlock model to address this problem. In such a system, the

repository allows only one person to change a file at a time.

First Harry must lock the file before he can begin making changes to it. Locking a file is a lot like borrowing a book from the library; if Harry has locked a file, then Sally cannot make any changes to it. If she tries to lock the file, the repository will deny the request.

All she can do is read the file, and wait for Harry to finish his changes and release his lock. After Harry unlocks the file, his turn is over, and now Sally can take her turn by locking and editing.

The problem with the lock-modify-unlock model is that it's a bit restrictive, and often becomes a roadblock for users:

- Locking may cause administrative problems. Sometimes Harry will lock a file and then forget about it. Meanwhile, because Sally is still waiting to edit the file, her hands are tied. And then Harry goes on vacation. Now Sally has to get an administrator to release Harry's lock. The situation ends up causing a lot of unnecessary delay and wasted time.
- Locking may cause unnecessary serialization. What if Harry is editing the begin- ning of a text file, and Sally simply wants to edit the end of the same file? These changes don't overlap at all. They could easily edit the file simultaneously, and no great harm would come, assuming the changes were properly merged together. There's no need for them to take turns in this situation.
- Locking may create a false sense of security. Pretend that Harry locks and edits file A, while Sally simultaneously locks and edits file B. But suppose that A and B depend on one another, and the changes made to each are semantically incom- patible. Suddenly A and B don't work together anymore. The locking system was powerless to prevent the problem yet it somehow provided a false sense of security. It's easy for Harry and Sally to imagine that by locking files, each is beginning a safe, insulated task, and thus not bother discussing their incompatible changes early on.

The Copy-Modify-Merge Solution

Most version control systems use a copy-modify-merge model as an alternative to locking. In this model, each user's client contacts the project reposi- tory and creates a personal working copya local reflection of the repository's files and directories. Users then work in parallel, modifying their private copies. Finally, the private copies are merged together into a new, final version. The version control system often assists with the merging, but ultimately a human being is responsible for making it happen correctly.

Here's an example. Say that Harry and Sally each create working copies of the same project, copied from the repository. They work concurrently, and make changes to the same file A within their copies. Sally saves her changes to the repository first.

When Harry attempts to save his changes later, the repository informs him that his file A is out-of-date. In other words, that file A in the repository has somehow changed since he last copied it. So Harry asks his client to merge any new changes from the repository into his working copy of file A.

Chances are that Sally's changes don't overlap with his own; so once he has both sets of changes integrated, he saves his working copy back to the repository.

But what if Sally's changes do overlap with Harry's changes? What then? This situation is called a conflict, and it's usually not much of a problem. When Harry asks his client to merge the latest repository changes into his working copy, his copy of file A is

somehow flagged as being in a state of conflict: he'll be able to see both sets of conflicting changes, and manually choose between them. Note that software can't automatically resolve conflicts; only humans are capable of understanding and making the necessary intelligent choices. Once Harry has manually resolved the overlapping changes, perhaps after a discussion with Sally, he can safely save the merged file back to the repository.

The copy-modify-merge model may sound a bit chaotic, but in practice, it runs extremely smoothly. Users can work in parallel, never waiting for one another. When they work on the same files, it turns out that most of their concurrent changes don't overlap at all; conflicts are infrequent. And the amount of time it takes to resolve conflicts is far less than the time lost by a locking system.

In the end, it all comes down to one critical factor: user communication. When users communicate poorly, both syntactic and semantic conflicts increase. No system can force users to communicate perfectly, and no system can detect semantic conflicts.

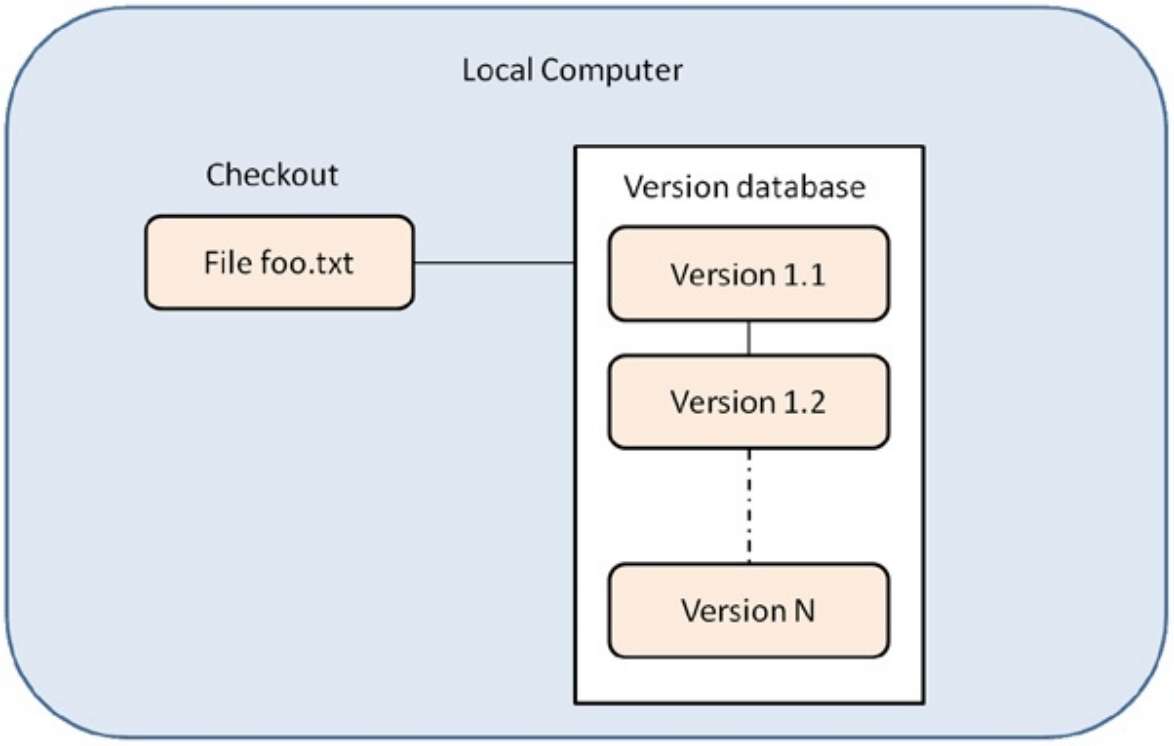
So there's no point in being lulled into a false promise that a locking system will somehow prevent conflicts; in practice, locking seems to inhibit productivity more than anything else.

Types of Version Control Systems

There are three types of version control systems available. These are classified based on their mode of operation:

- Local version control system
- Centralized version control system
- Distributed version control system

Local version control system

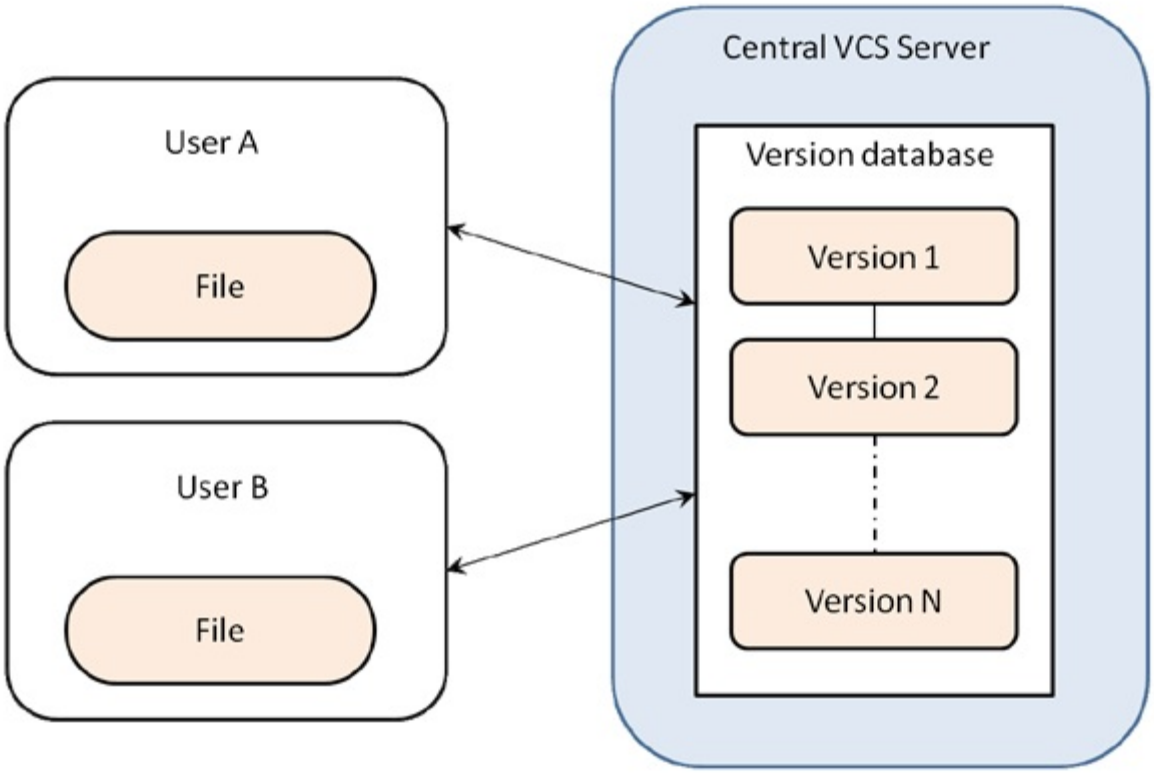


A local version control system is the first successful attempt to solve the file sharing issue.

Local version control system works by keeping patch sets (that is, the difference between the file's content at progressive stages) using a special format in the version tracker that is stored in your local hard disk.

It can then recreate the file's contents exactly at any given point in time by adding up all the relevant patches in order and "checking it out" (reproducing the content to the user's workplace). This only works on a single computer and is not accessible to other people working on the same files

Centralized Version Control system



Central Version Control keeps the files in a common place (server) that everybody has access to from their local machines (clients).

Whenever people want to edit single or multiple files only the last version of the files are retrieved.

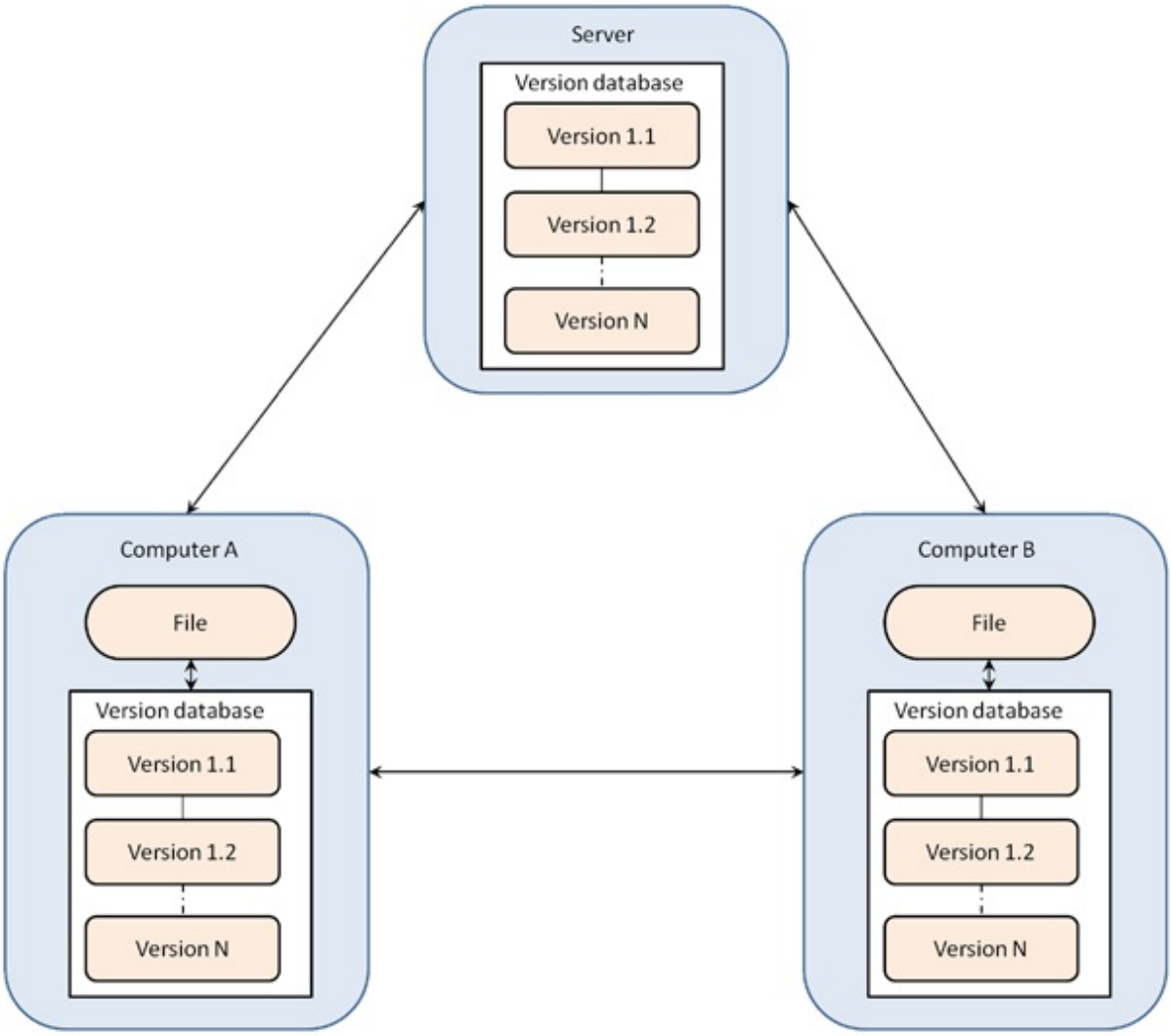
This setup not only provides access to the files for people who require them but also offers visibility on what other people are working towards.

As the files are stored in one single location from which everybody needs to share the files, any changes made to the files are automatically shared with other individuals as well.

The dangers of this system is that, on one single unit, the probability of losing is high.

There is a high degree of risk involved in using a centralized version control system because the users only have the last version of files in their system for working purposes; there is a chance one might ultimately lose the entire history of your files if the server gets corrupt.

Distributed version control system



Distributed version control systems are meant to address the short comming of centralised systems. It is a hybrid of the local and centralised system.

A distributed version control system is designed to store the entire history of the file/files on each and every machine locally and also syncs the local changes made by the user back to the server whenever required so that the changes can be shared with others providing a collaborative working environment.

Distributed version control systems have the advantages of local version control systems, such as the following:

- Making local changes without any concern of full time connectivity to the server
- Not relying on a single copy of files stored in the server
- Reusability of work
- Collaborative working, not relying on history stored on individual machines

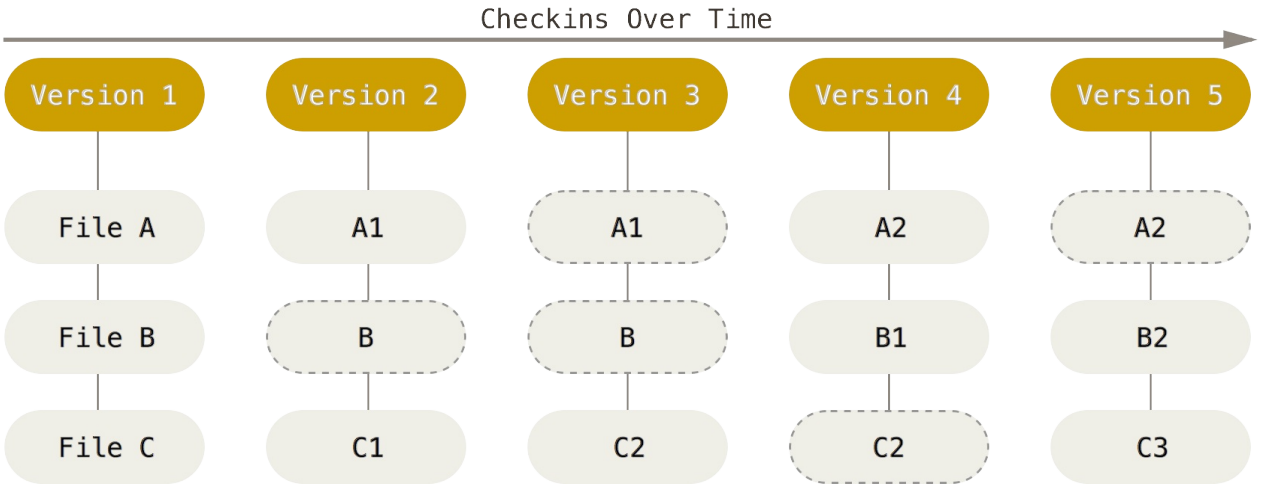
There are many version control systems such as Subversion, Mercurial, CVS and Git.

This course will cover Git, a Distributed version control system in depth

Git Version Control Basics

The idea behind Git to keep snapshots of your coding projects. This means that, as you code, at intervals you declare a point in the history of the project.

Git treats your of its data more like a set of snapshots of a mini filesystem. Every time you commit, or save the state of your project in Git, it basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot. If files have not changed, Git doesn't store the file again, just a link to the previous identical file it has already stored. As show in the figure below



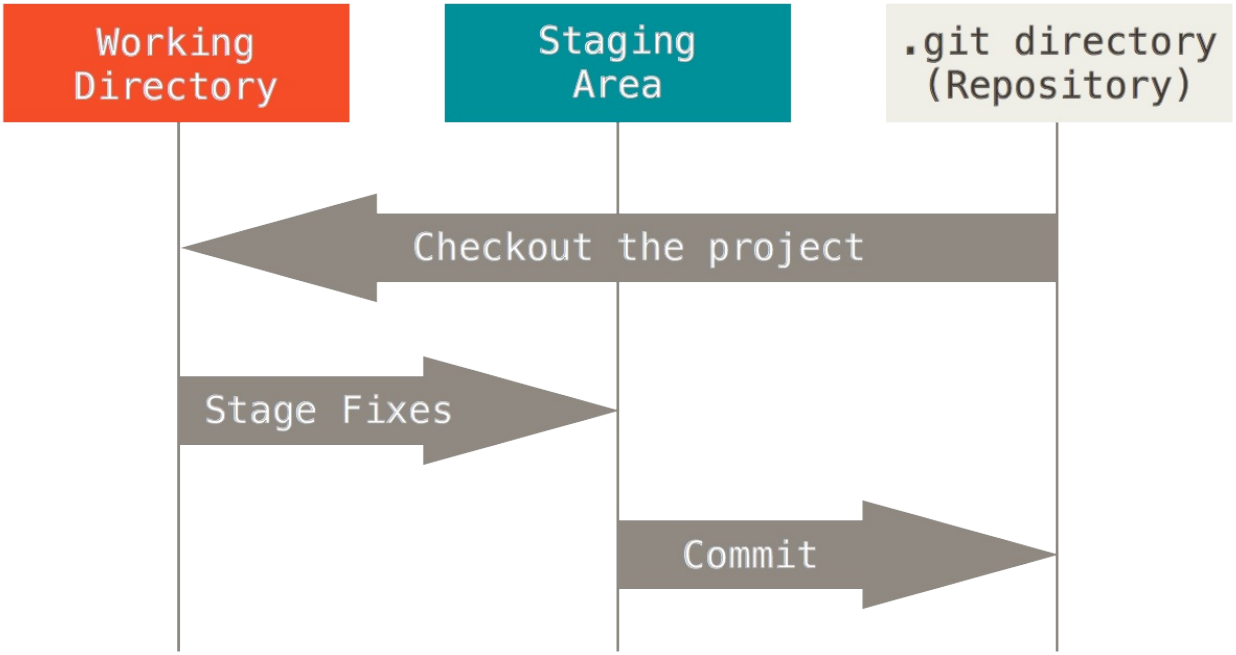
Git is Local. Most operations in Git only need local files and resources to operate – generally no information is needed from another computer on your network.

Git has Three States

Git has three main states that your files can reside in: **committed**, **modified**, and **staged.**

- Committed:Data is safely stored in your local database.
- Modified: You have changed the file but have not committed it to your database yet.
- Staged: You have marked a modified file in its current version to go into your next commit snapshot.

This leads us to the three main sections of a Git project: the Git directory, the working directory, and the staging area.



The Git directory is is for the metadata and object database for the project.It is what is copied when you clone a repository from another computer.

The working directory is a single checkout of one version of the project.Files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify.

The staging area is a file, generally contained in your Git directory, that stores information about what will go into your next commit (Index) .

The basic Git workflow goes something like this:

1. You modify files in your working directory.
2. You stage the files, adding snapshots of them to your staging area.
3. You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

Git Version Control Hands on

Identity

The commands below sets your identity

```
$ git config global user.name "John Banda"
$ git config global user.email johbbanda@example.com
```

Setting Git Editor on the CLI

Command below sets your Editor as vim on linux you can set pico , gedit or emacs

```
$ git config --global core.editor vim
```

Listing your Settings

You can list Git information for the account before you start working. This command is very handy since you will be working on shared machines. Use the confing command to change settings if they do not reflect your identity

```
git config --list
```

Help Commands

There are three ways to get the manual page (manpage) help for any of the Git commands:

```
$ git help <verb>
$ git <verb> --help
$ man git-<verb>
```

Creating a Git Repository

You create a Git project using two main approaches. The first one is to take an existing project or directory and imports it into Git. The second clones an existing Git repository from another server.

Repo from Scratch

If you're starting with a new project , you need to go to the project's directory and type

```
$ git init
```

This creates a new subdirectory named .git that contains all of your necessary repository files – a Git repository skeleton.

At this point nothing is tracked by Git. To begin the tracking. To begin the tracking use the Git **add** command that specify the files you want to track, followed by a git commit as show below

```
$ touch test1.txt test2.txt
$ git add *.txt
$ git add LICENSE
$ git commit -m 'initial project version'
```

Cloning an Existing Repo

If you want to get a copy of an existing Git repository the command you need is **git clone** .

You clone a repository with git clone [url] .

```
$ git clone https://github.com/boniface/cput.git
```

This creates a directory named "cput", initializes a .git directory inside it, pulls down all the data for that repository, and checks out a working copy of the latest version.

If you go into the new cput directory, you'll see the project files in there, ready to be worked on or used. If you want to clone the repository into a directory named something other than "cput", you can specify that as the next command-line option:

```
$ git clone https://github.com/boniface/cput.git myproject
```

That command does the same thing as the previous one, but the target di- rectory is called myproject

Tracking Changes in the Repository

Each file in your working directory can be in one of two states: Tracked or Untracked.

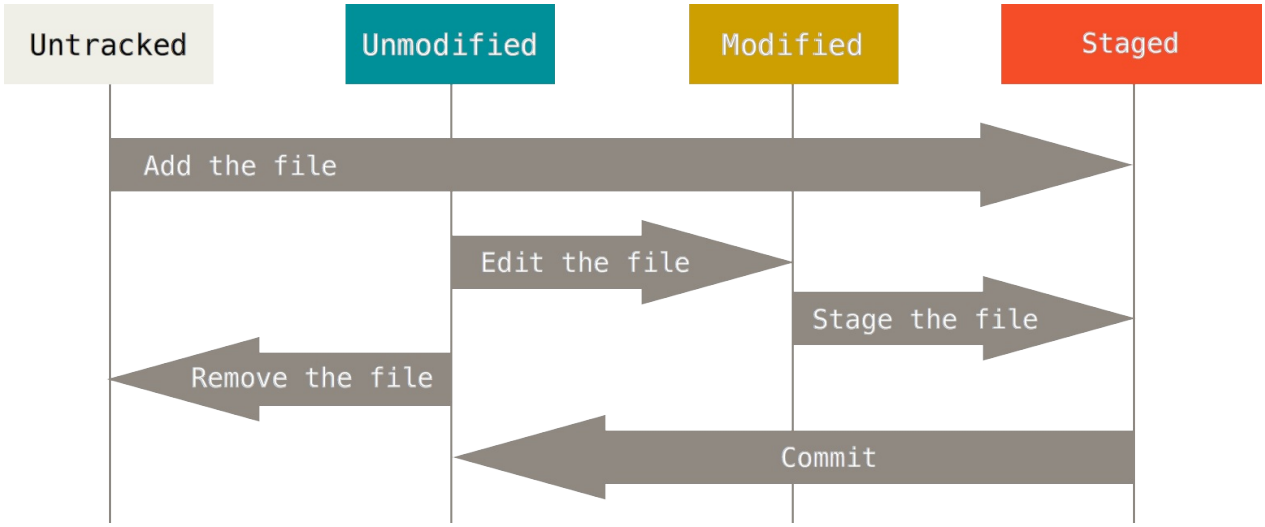
Tracked files are files that were in the last snapshot; they can be unmodified, modified, or staged.

Untracked files are everything else – any files in your working directory that were not in your last snapshot and are not in your staging area.

When you first clone a repository, all of your files will be tracked and unmodified because you just checked them out and haven't edited anything.

As you edit files, Git sees them as modified, because you've changed them since your last commit.

You stage these modified files and then commit all your staged changes, and the cycle repeats.



Checking Files Status

The main tool you use to determine which files are in which state is the **git status** command. If you run this command directly after a clone, you should see information about the files:

```
$ git status
```

To ee changes run the command below

```
$ echo 'My Project' > README
$ git status
```

You should see that your new README file is untracked, but not added to your repository.

Tracking New File

In order to begin tracking a new file, you use the command git add . To begin tracking the README file, you can run this:

```
$ git add README
```

And check Status

```
$ git status
```

Short Status

Git has a short status flag so you can see your changes in a more compact way. If you run git status -s or git status --short you get a far more simpli- fied output from the command.

```
$ git status -s
```

Ignoring Files

If you have a class of files that you don't want Git to automatically add or even show you as being untracked, you can tell Git to ingnore them. These are generally automatically generated files such as log files or files produced by your build system. In such cases,you can create a file listing patterns to match them named .gitignore . Here is an example .gitignore file:

You can view the contents as

```
$ cat .gitignore
```

You can put the following entries n your file to get the commented effect

```
# no .a files
*.a
# but do track lib.a, even though you're ignoring .a files above
!lib.a
# only ignore the root TODO file, not subdir/TODO

/TODO
# ignore all files in the build/ directory
build/
# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt
# ignore all .txt files in the doc/ directory
doc/**/*.txt
```

For more examples of .gitinore go to https://github.com/github/ gitignore

Viewing Staged and Unstaged Changes

There are commands to view What you have changed but not yet staged and what you you staged that you are about to commit.

Although git status answers those questions very generally by listing the file names, git diff shows you the exact lines added and removed often called the patch.

```
$ git diff
```

The command compares what is in your working directory with what is in your staging area. The result tells you the changes you've made that you ha- ven't yet staged. If you want to see what you've staged that will go into your next commit, you can use git diff --staged . This command compares your staged changes to your last commit:

```
$ git diff --stagged
```

Note that **git diff** by itself doesn't show all changes made since your last commit – only changes that are still unstaged.

git diff --cached is used to see what you've staged so far:

```
$ git diff --cached
```

Committing Changes

Git only committs stage files. The simplest way to com- mit is to type git commit :

```
$ git commit
```

This launches your editor of choice in this case vim r you to write a message

Alternatively, you can type your commit message inline with the commit command by specifying it after a -m flag, like this:

```
$ git commit -m "This is my First Committ" Please write meaning message of what you have done
```

You can skip the staging stage, that is doing git add by using the command below:

```
$ git commit -a -m 'Write message aof what you have done'
```

Removing Files

To remove a file from Git, you have to remove it from your tracked files (your staging area) and then commit. The git rm command does that, and also removes the file from your working directory so you don't see it as an untracked file the next time around.

```
$ git rm README
```

If you simply remove the file from your working directory, it shows up under the “Changed but not updated” (that is, unstaged) area of your git status output

```
$ rm README
```

The next time you commit, the file will be gone and no longer tracked.

Viewing the Commit History

After you have created several commits, or if you have cloned a repository with an existing commit history, you’ll probably want to look back to see what has happened.

For that you use

```
$ git log
```

Undoing Things

One of the common undos takes place when you commit too early and pos- sibly forget to add some files, or you mess up your commit message. If you want to try that commit again, you can run commit with the --amend option:

```
$ git commit -m 'initial commit'
$ git add forgotten_file
$ git commit --amend
```

You end up with a single commit – the second commit replaces the results of the first.

Unstaging a Staged File

```
$ git reset HEAD README
```

Unmodifying a Modified File

If you realize that you don’t want to keep your changes to a file, you can easily unmodify it – revert it back to what it looked like when you last committed

```
$ git checkout -- README
```

Working with Remotes Repositories

Remote repositories are versions of your project that are hosted on the Internet or network somewhere

To see which remote servers you have configured, you can run the git remote command

```
$ git clone https://github.com/boniface/cput.git myproject
$ cd myproject
$ git remote
$ git remote -v
```

You can also specify -v , which shows you the URLs that Git has stored for the shortname to be used when reading and writing to that remote. If you have more than one remote, the command lists them all

Adding Remote Repositories

To add a new remote Git repository as a shortname you can reference easily, run git remote add [shortname] [url] :

```
$ git remote
$ git remote add pb https://github.com/boniface/cput.git
$ git remote -v
origin https://github.com/paulboone/ticgit (fetch)
orrigin https://github.com/paulboone/ticgit (push)
```

Fetching and Pulling from Your Remotes

To get data from your remote projects, you can run git fetch [remote-name]

Fetch

```
$ git fetch origin
```

The command goes out to that remote project and pulls down all the data from that remote project that you don’t have yet. After you do this, you should have references to all the branches from that remote, which you can merge in or inspect at any time

If you clone a repository, the command automatically adds that remote repository under the name “origin”.

git fetch origin fetches any new work that has been pushed to that server since you cloned (or last fetched from) it

It’s important to note that the git fetch command pulls the data to your local repository – it doesn’t automatically merge it with any of your work or modify what you’re currently working on. You have to merge it manually into your work when you’re ready.

Pull

If you have a branch set up to track a remote branch, you can use the git pull command to automatically fetch and then merge a remote branch into your current branch.

```
$ git pull  [remote-name]
```

The git clone command automatically sets up your local master branch to track the remote master branch (or whatever the default branch is called) on the server you cloned from. Running git pull generally fetches data from the server you originally cloned from and automatically tries to merge it into the code you’re currently working on.

Pushing to Remotes

When you have your project at a point that you want to share, you have to push it upstream. The command for this is simple: git push [remote-name]

If you want to push your master branch to your origin server then you can run this to push any commits you've done back up to the server:

```
$ git push origin master
```

This command works only if you cloned from a server to which you have write access and if nobody has pushed in the meantime. If you and someone else clone at the same time and they push upstream and then you push upstream, your push will rightly be rejected. You'll have to pull down their work first and incorporate it into yours before you'll be allowed to push.

Inspecting a Remote

f you want to see more information about a particular remote, you can use the git remote show [remote-name] command.

```
$ git remote show origin
```

Removing and Renaming Remotes

If you want to rename a reference you can run git remote rename to change a remote's shortname. For instance, if you want to rename pb to paul , you can do so with git remote rename :

```
$ git remote rename jk jackal
$ git remote
```

Tagging

Tagging is used to name your important mark points for your ce base. In this section, you'll learn how to list the available tags, how to create new tags, and what the different types of tags are.

Listing Tags

Listing the available tags in Git is straightforward. Just type git tag :

```
$ git tag
```

The commands follows the linux or unix ls command with all options

Creating Tags

Git uses two main types of tags: lightweight and annotated.

A lightweight tag is very much like a branch that doesn't change – it's just a pointer to a specific commit.

Annotated tags, however, are stored as full objects in the Git database. They're checksummed; contain the tagger name, e-mail, and date; have a tag- ging message; and can be signed and verified.

It's generally recommended that you create annotated tags so you can have all this information

Creating an annotated tag in Git is simple. The easiest way is to specify -a when you run the tag command:

```
$ git tag -a v0.1 -m ' Intitial working Version'
$ git tag
```

The -m specifies a tagging message, which is stored with the tag.

Tagging Later

You can also tag commits after you've moved past them. suppose you forgot to tag the project at v1.2, which was at the "upda- ted rakefile" commit. You can add it after the fact. To tag that commit, you spec- ify the commit checksum (or part of it) at the end of the command:

```
git tag -a v1.2 9fceb02
```

Sharing Tags

By default, the git push command doesn't transfer tags to remote servers. You will have to explicitly push tags to a shared server after you have created them. This process is just like sharing remote branches – you can run git push origin [tagname] .

```
git push origin v1.5
```

If you have a lot of tags that you want to push up at once, you can also use the --tags option to the git push command. This will transfer all of your tags to the remote server that are not already there.

```
git git push origin --tags
```

Checking out Tags

You can't check out a tag in Git, since they can't be moved around. If you want to put a version of your repository in your working directory that looks like a specific tag, you can create a new branch at a specific tag:

```
git checkout -b version2 v2.0.0
```

Git Version Control Branching

Nearly every Version Control Systems have branching support. Branching means you diverge from the main line of development and continue to do work without messing with that main line. Git encourages workflows that branch and merge often, even multiple times in a day.

Git stores ddata as a series of snapshots. When you make a commit, Git stores a commit object that contains a pointer to the snapshot of the content you staged. This object also contains the author's name and email, the message that you typed, and pointers to the commit or commits that directly came before this commit : zero parents for the initial commit, one parent for a normal commit, and multiple parents for a commit that results from a merge of two or more branches.

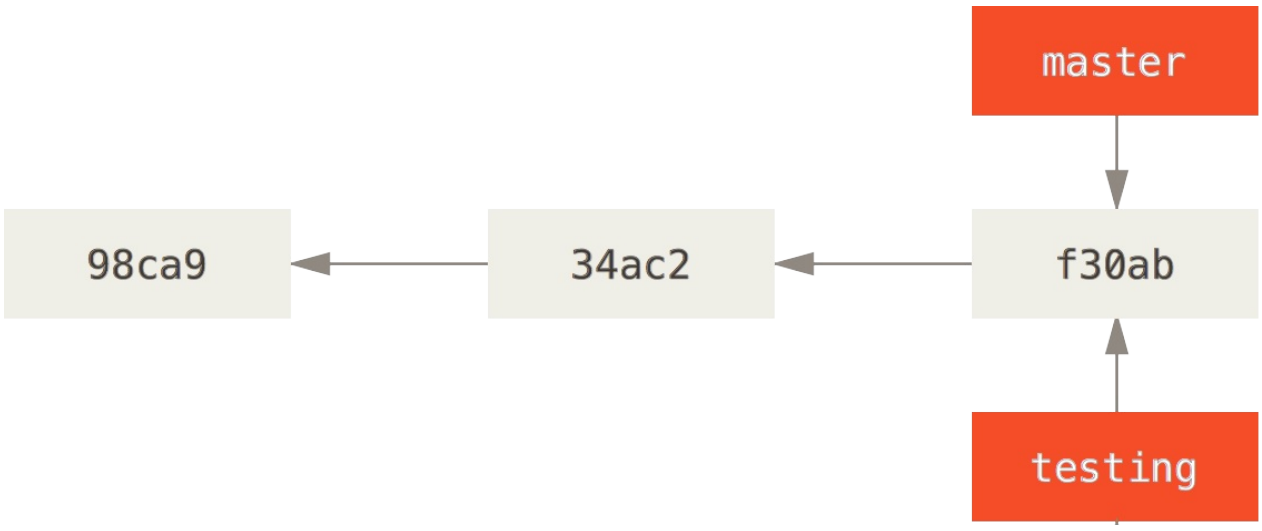
A branch in Git is simply a lightweight movable pointer to one of these commits. The default branch name in Git is **master** and a lot of people don't bother to change it. As you start making commits, you're given a master branch that points to the last commit you made. Every time you commit, it moves forward automatically.

Creating a New Branch

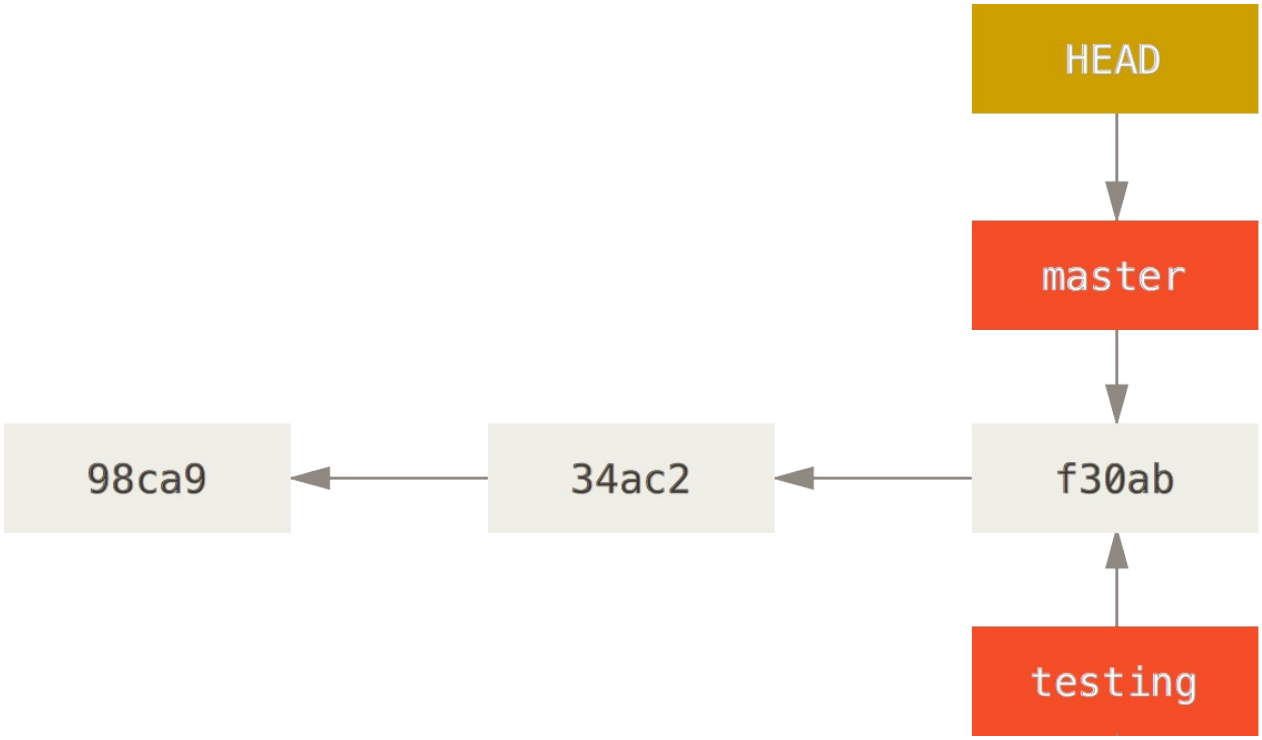
Creating a new branch just creates a a new pointer for you to move around. Let's say you create a new branch called **testing**, you have just created a new pointer in addition to the **master pointe**r and you do this with the git branch command:

```
$ git branch testing
```

This creates a new pointer at the same commit you're currently on as shown below



Git knows what branch you're currently by keeping a special pointer called HEAD . This is a pointer to the local branch you're currently on as show below . In this case, you're still on master. The git branch command only created a new branch – it didn't switch to that branch.



You can easily see this by running a simple git log command that shows you where the branch pointers are pointing. This option is called --decorate .

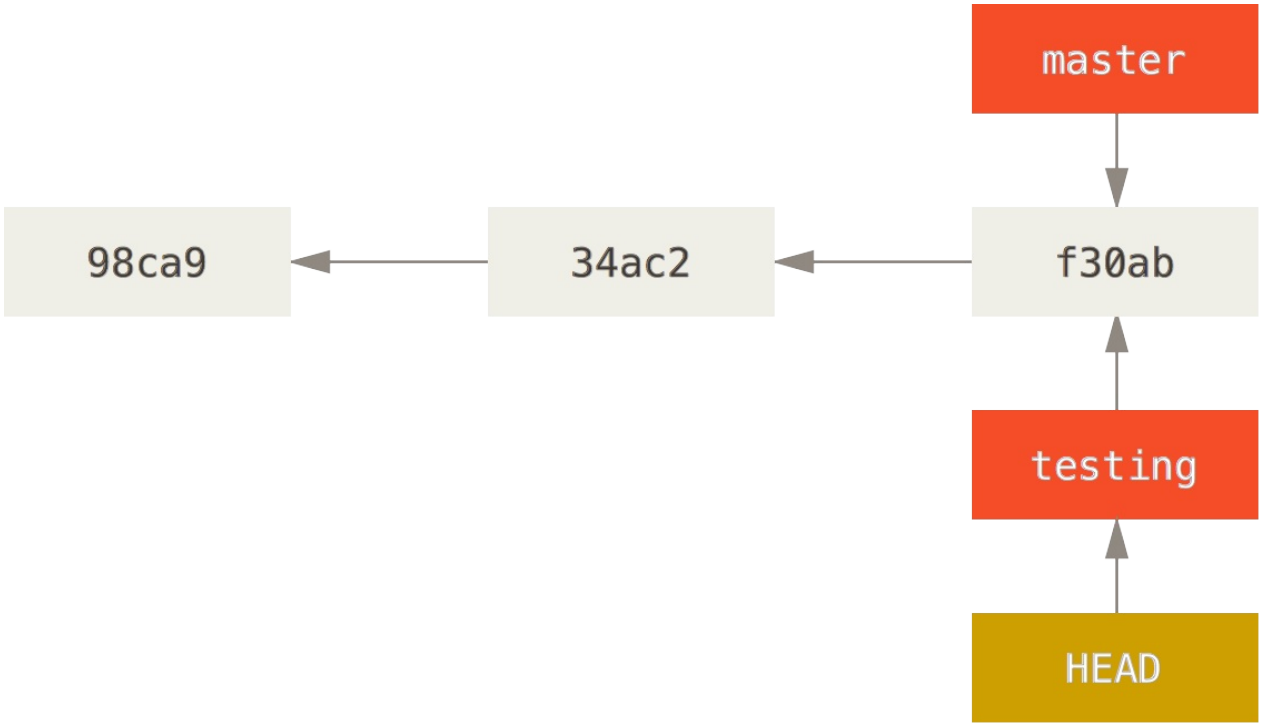
```
$ git log --oneline --decorate
```

Switching Branches

To switch to an existing branch, you run the git checkout command. Let's switch to the new testing branch:

```
$ git checkout testing
```

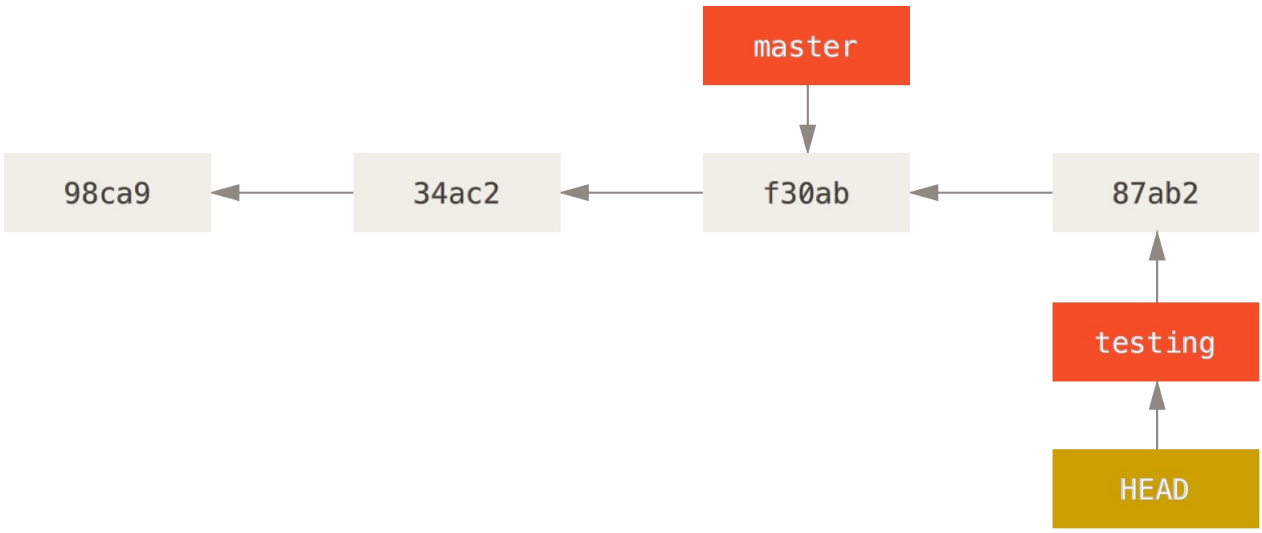
This moves HEAD to point to the testing branch as show below



Issuing a commit at this stage moves the testing branch forward with the new snapshort

```
$ touch test.java
$ git commit -a -m 'made a change'
```

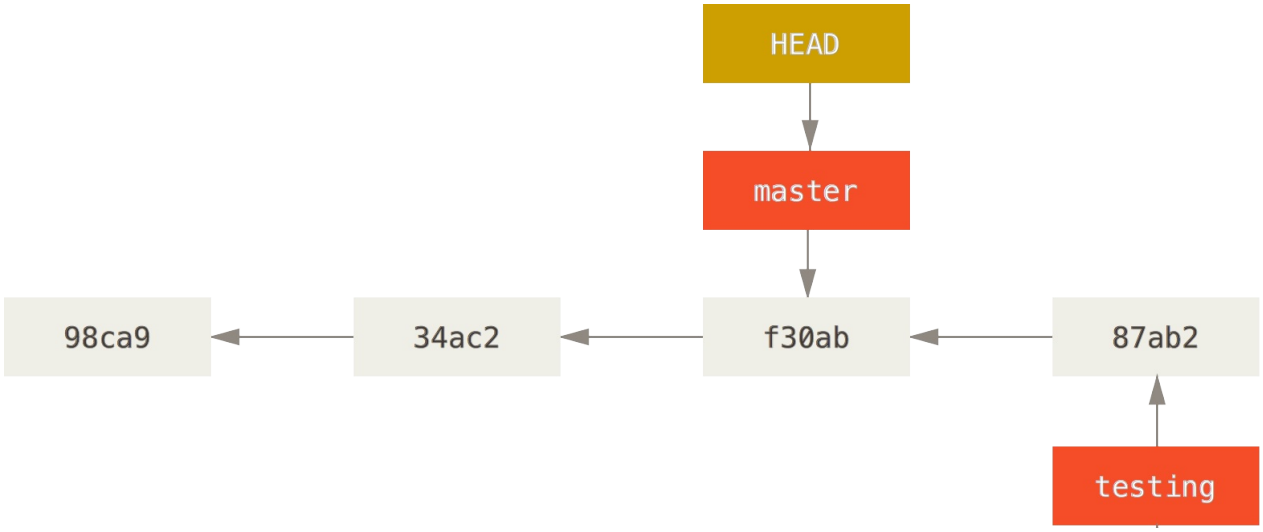
The snapshort now looks like the diagram below



The testing branch has moved forward, but the master branch still points to the commit you were on when you ran git checkout to switch branches. To switch back to master , run the checkout command for master

```
$ git checkout master
```

The diagram now looks like what is shown below

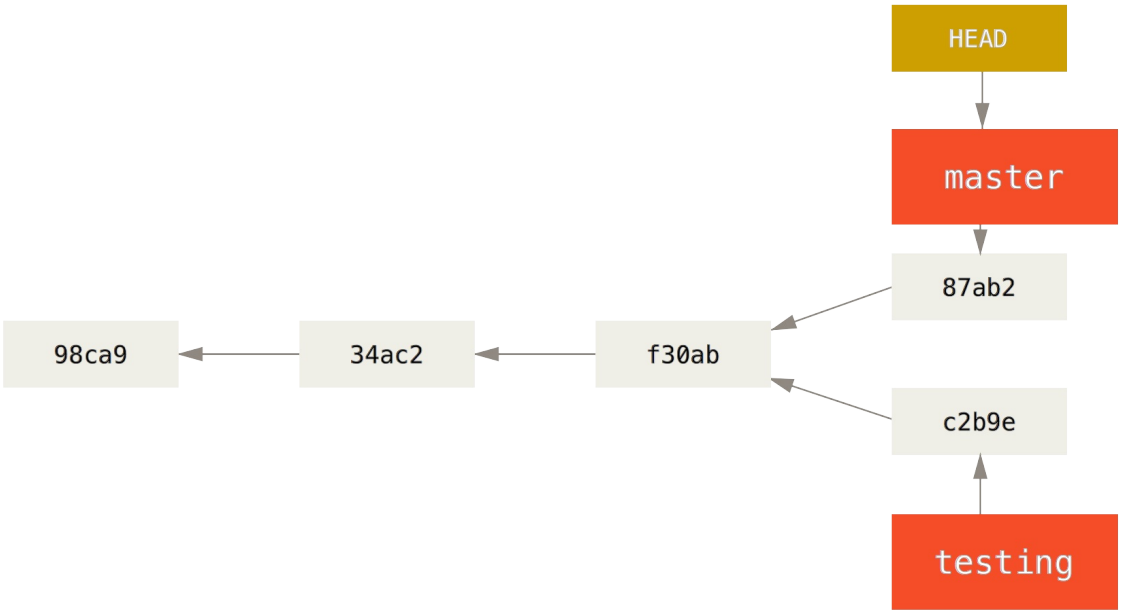


The checkout command did two things: It moved the HEAD pointer back to point to the master branch, and it reverted the files in your working directory back to the snapshot that master points to. This also means the changes you make from this point forward will diverge from an older version of the project. It es- sentially rewinds the work you've done in your testing branch so you can go in a different direction.

Let's make a few changes and commit again:

```
$ touch test.java
$ git commit -a -m 'made a change'
```

Now the project history has diverged. You created and switched to a branch, did some work on it, and then switched back to your main branch and did other work. Both of those changes are isolated in separate branches: you can switch back and forth between the branches and merge them together when you're ready. And you did all that with simple branch , checkout , and commit commands.



You can also see this easily with the git log command. If you run git log --oneline --decorate --graph --all it will print out the history of your commits, showing where your branch pointers are and how your history has di- verged.

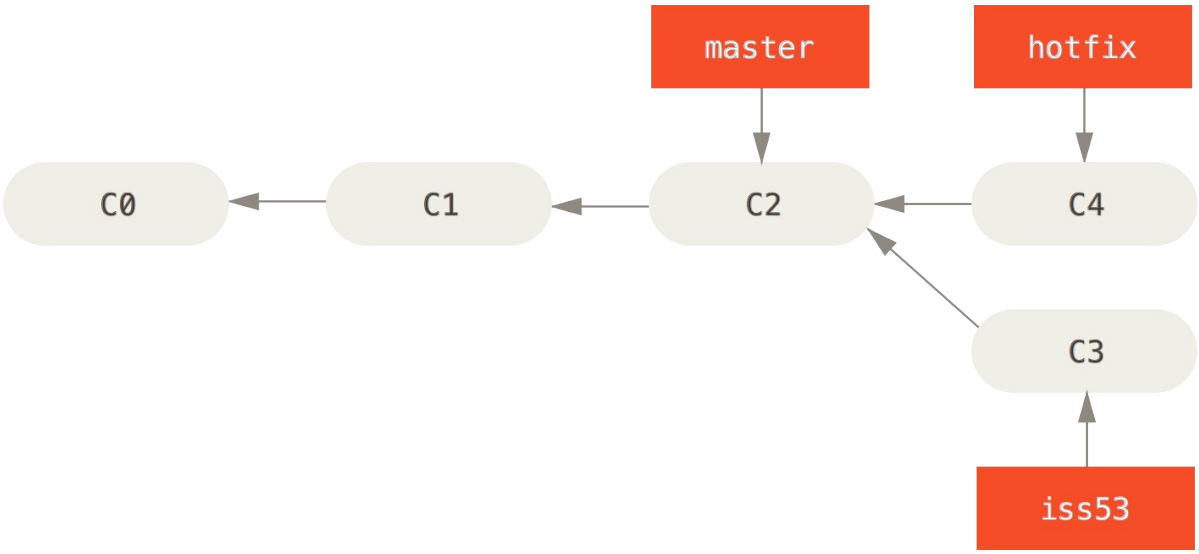
```
$ git log --oneline --decorate --graph --all
```

Merging

Lets assume you have a branch that you have created using the command below

```
$ git branch iss53
$ git checkout iss53
$ git checkout master
$ git checkout -b hotfix
$ touch test.java
$ git commit -a -m 'Added a Hot Fix'
```

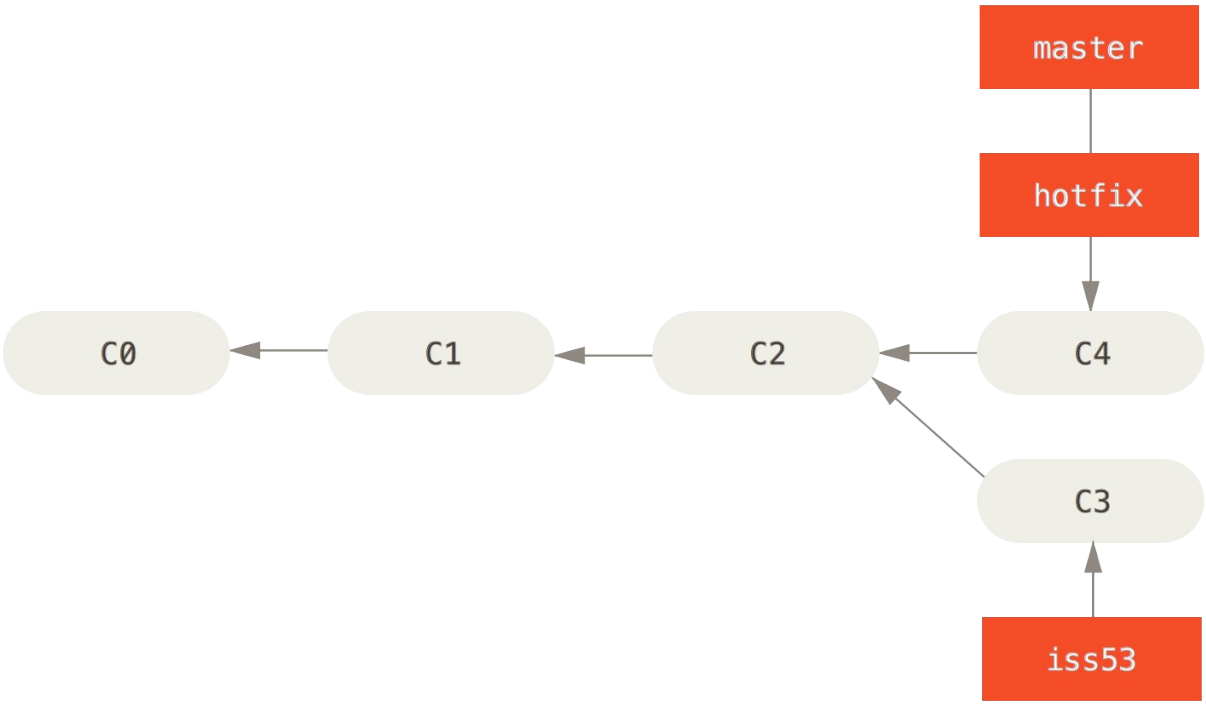
And it looks like the diagram below



You van merge the Hotfix back into your master branch the git merge command:

```
$ git checkout master
$ git merge hotfix
```

The new snapshot will now look like below



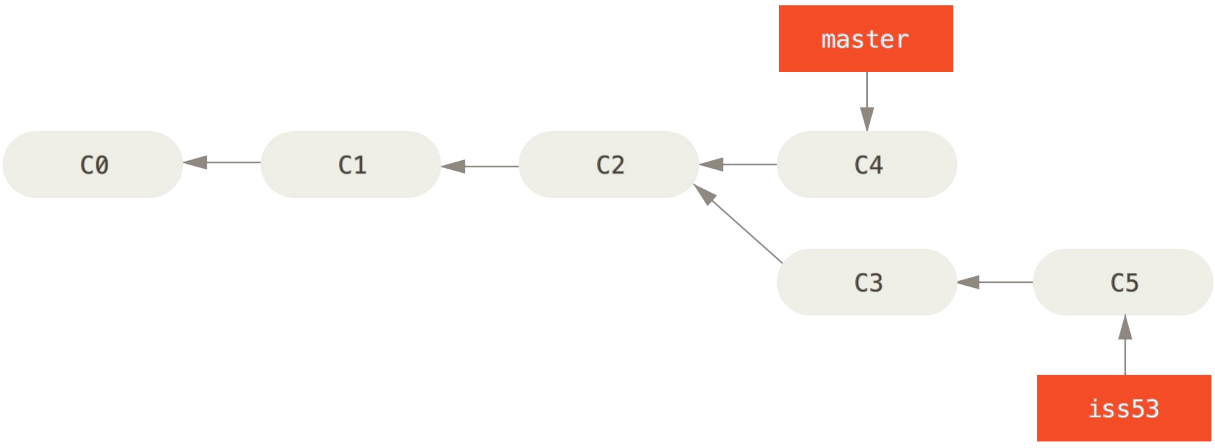
After you have merged the hotfix branch into the master branch, ou'll delete the hotfix branch, because you no longer need it – the master branch points at the same place. You can delete it with the -d option to git branch :

```
$ git branch -d hotfix
```

At this point you can move to another branch **iss53** with command

```
$ git checkout iss53
$ touch test1.java
$ git commit -a -m 'finished branch 53'
```

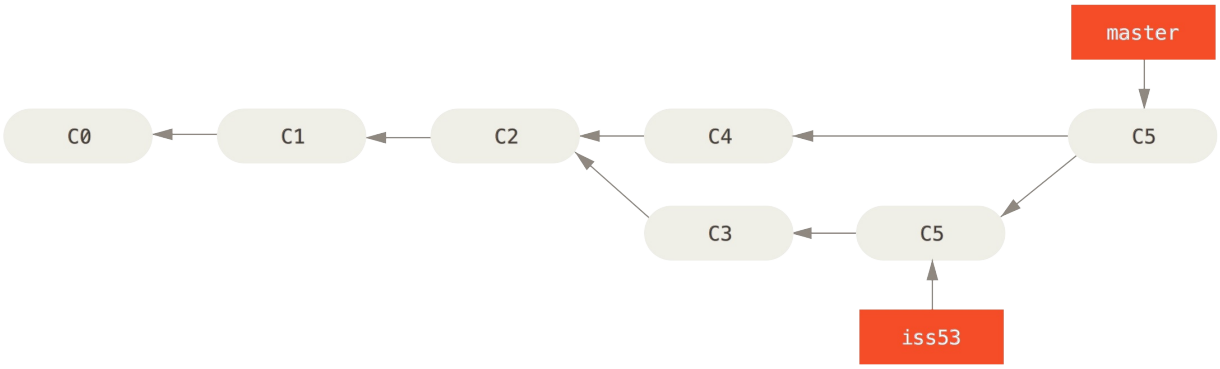
And the snapshort will look as below



You can now merge the iss53 into the master with the commands below

```
$ git checkout master
$ git merge iss53
```

The snapshot will look like the diagram below



Now that your work is merged in, you have no further need for the iss53 branch. You can delete the branch:

```
$ git branch -d iss53
```

Merge Conflicts

If you changed the same part of the same file differently in the two branches you're merging together, Git won't be able to merge them cleanly. If your fix for issue #53 modified the same part of a file as the hotfix , you'll get a merge conflict message.

If you want to see which files are unmerged at any point after a merge conflict, you can run git status

Remote Branches Revisited

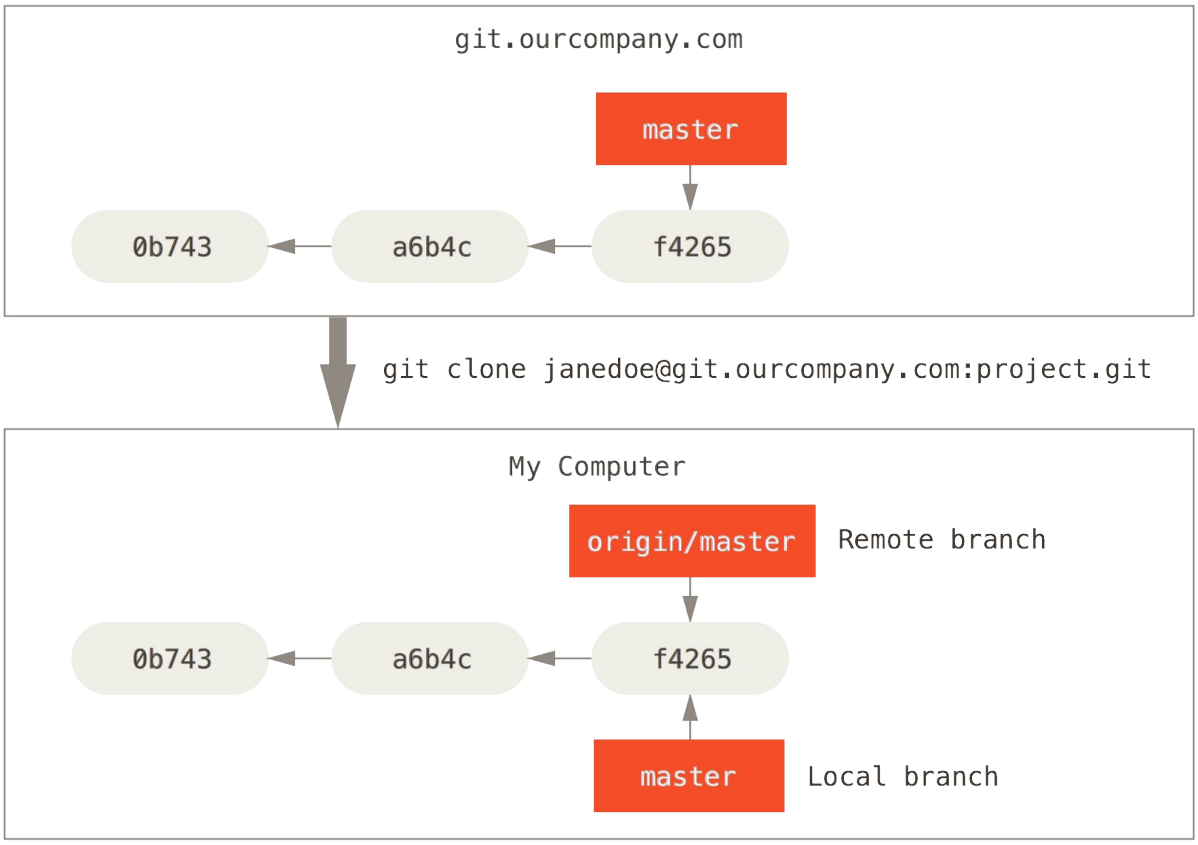
Remote branches are references to the state of branches in your remote repositories. They're local branches that you can't move; they're moved automatically for you whenever you do any network communication.

Remote branches act as bookmarks to remind you where the branches on your remote repositories were the last time you connected to them.

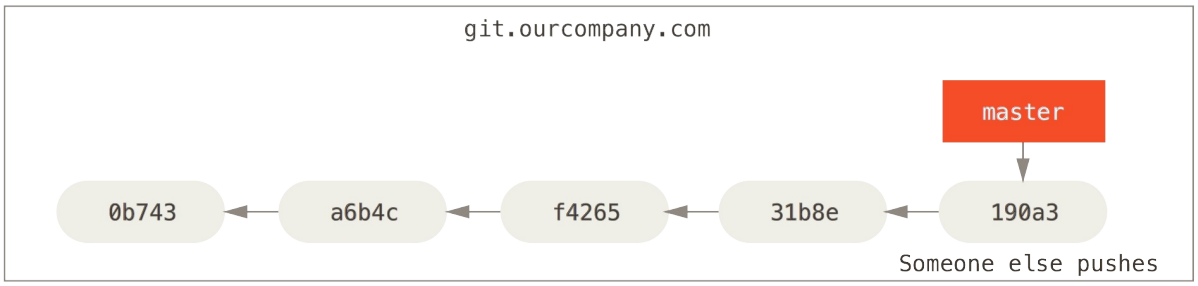
They take the form (remote)/(branch) . For instance, if you wanted to see what the master branch on your origin remote looked like as of the last time you communicated with it, you would check the origin/master branch.

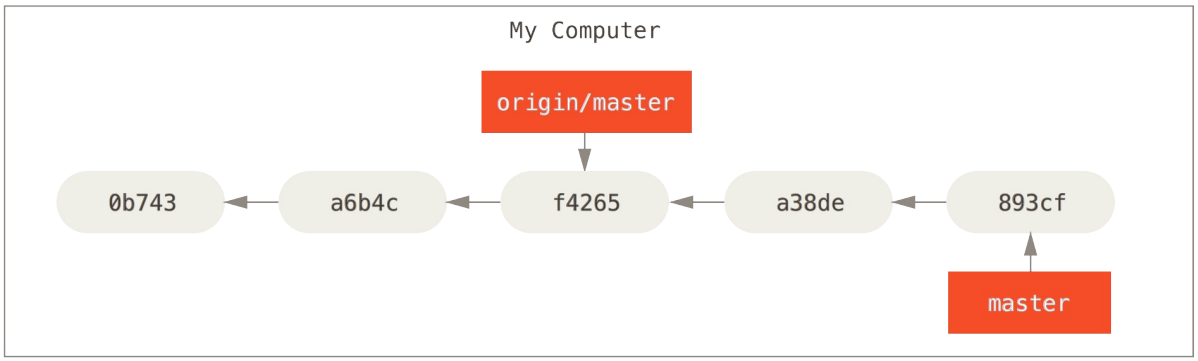
If you were working on an issue with a partner and they pushed up an iss53 branch, you might have your own local iss53 branch; but the branch on the server would point to the commit at origin/iss53 .

Let's say you have a Git server on your network at git.ourcompany.com . If you clone from this, Git's clone command automatically names it origin for you, pulls down all its data, creates a pointer to where its master branch is, and names it origin/ master locally. Git also gives you your own local master branch starting at the same place as origin's master branch, so you have something to work from. And belw is that snapshort

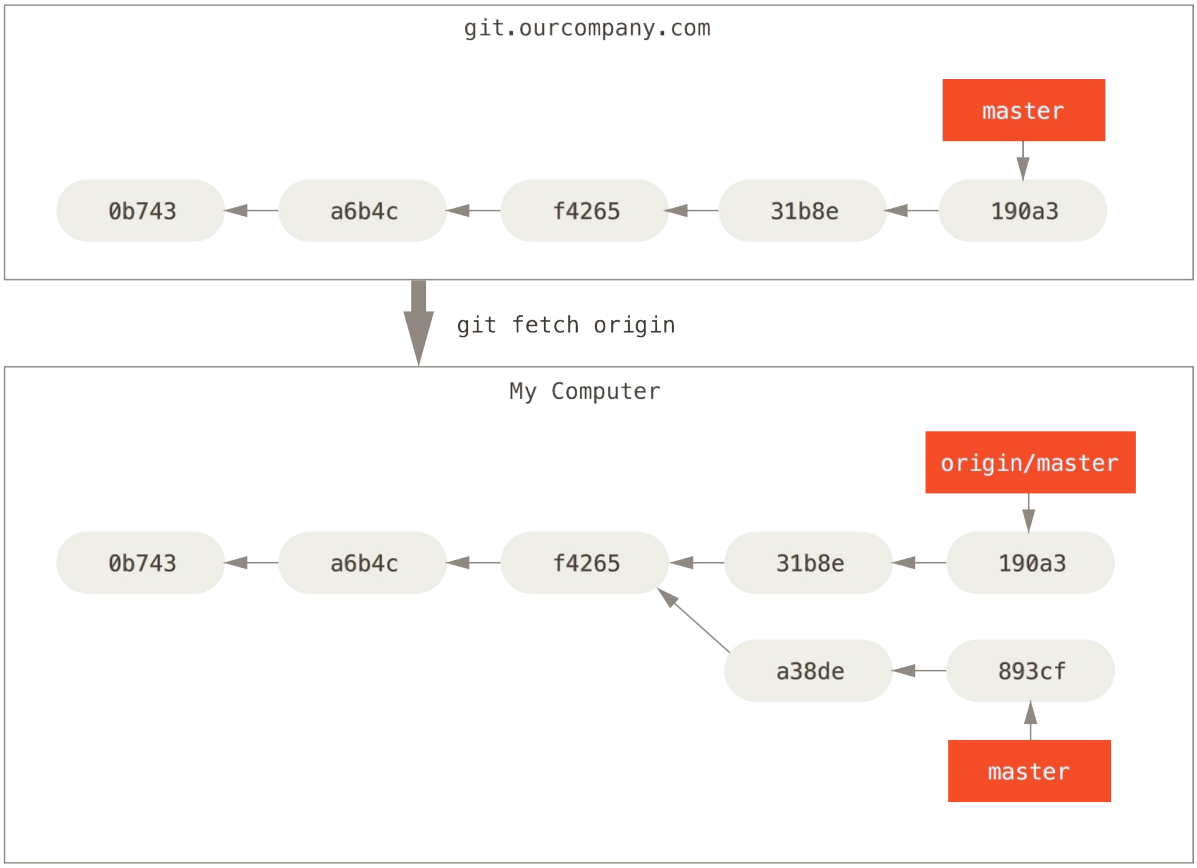


If you do some work on your local master branch, and, in the meantime, someone else pushes to git.ourcompany.com and updates its master branch, then your histories move forward differently. Also, as long as you stay out of contact with your origin server, your origin/master pointer doesn't move.





To synchronize your work, you run a git fetch origin command. This command looks up which server "origin" is (in this case, it's git.ourcompa- ny.com), fetches any data from it that you don't yet have, and updates your local database, moving your origin/master pointer to its new, more up-to-date position as show below



While the **git fetch** command will fetch down all the changes on the server that you don't have yet, it will not modify your working directory at all. It will simply get the data for you and let you merge it yourself.

However, there is a command called **git pull** which is essentially a git fetch immediately fol- lowed by a git merge.

Pushing

When you want to share a branch with the world, you need to push it up to a remote that you have write access to. Your local branches aren't automatically synchronized to the remotes you write to – you have to explicitly push the branches you want to share.

That way, you can use private branches for work you don't want to share, and push up only the topic branches you want to collaborate on.

If you have a branch named **hotfix** that you want to work on with others, you can push it up the same way you pushed your first branch. Run git push (remote) (branch)

```
$ git push origin hostfix
```

The next time one of your collaborators fetches from the server, they will get a reference to where the server's version of **hotfix** is under the remote branch origin/hotfix with the command

```
$ git fetch origin
```

Note that when you do a fetch that brings down new remote branches, you don't automatically have local, editable copies of them. In other words, in this case, you don't have a new hotfix branch – you only have an origin/hotfix

pointer that you can't modify.

To merge this work into your current working branch, you can run git merge origin/hotfix . If you want your own hotfix branch that you can work on, you can base it off your remote branch:

```
$ git checkout -b hotfix origin/hotfix
```

This gives you a local branch that you can work on that starts where origin/hotfix is.

Tracking Branches

Checking out a local branch from a remote branch automatically creates what is called a "upstream branch" or "tracking branch". Tracking branches are local branches that have a direct relationship to a remote branch.

If you're on a tracking branch and type git pull , Git automatically knows which server to fetch from and branch to merge into.

When you clone a repository, it generally automatically creates a master branch that tracks origin/master .

However, you can set up other tracking branches if you wish – ones that track branches on other remotes, or don't track the master branch. The simple case is the example you just saw, running git checkout -b [branch] [remotename]/[branch] .

```
$ git checkout --track origin/hotfix
```

To set up a local branch with a different name than the remote branch, you can easily use the first version with a different local branch name:

```
$ git checkout -b newbranchname origin/hotfix
```

Now, your local branch sf will automatically pull from origin/hostfix .

Deleting Remote Branches

Suppose you're done with a remote branch – say you and your collaborators are finished with a feature and have merged it into your remote's master branch (or whatever branch your stable codeline is in). You can delete a remote branch using the --delete option to git push . If you want to delete your serverfix branch from the server, you run the following:

```
$ git push origin --delete hotfix
```

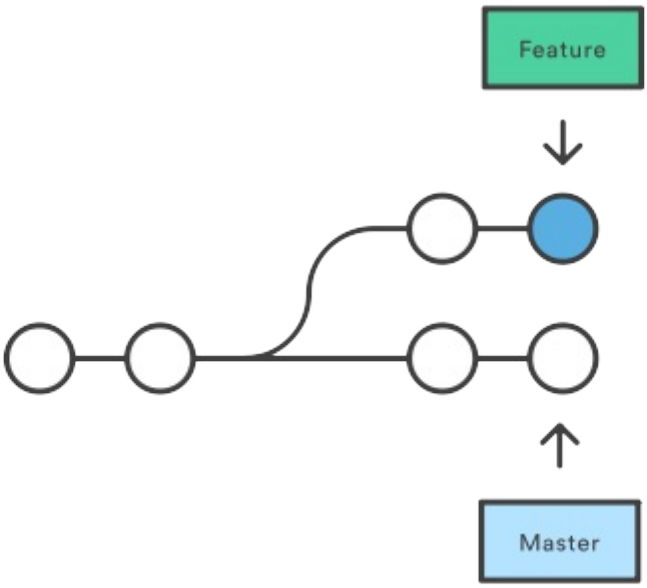
Rebasing

In Git, there are two main ways to integrate changes from one branch into an- other: the merge and the rebase . In this section you'll learn what rebasing is, how to do it and in what cases you won't want to use it.

Rebasing is the process of moving a branch to a new base commit. From a content perspective, rebasing really is just moving a branch from one commit to another. But internally, Git accomplishes this by creating new commits and

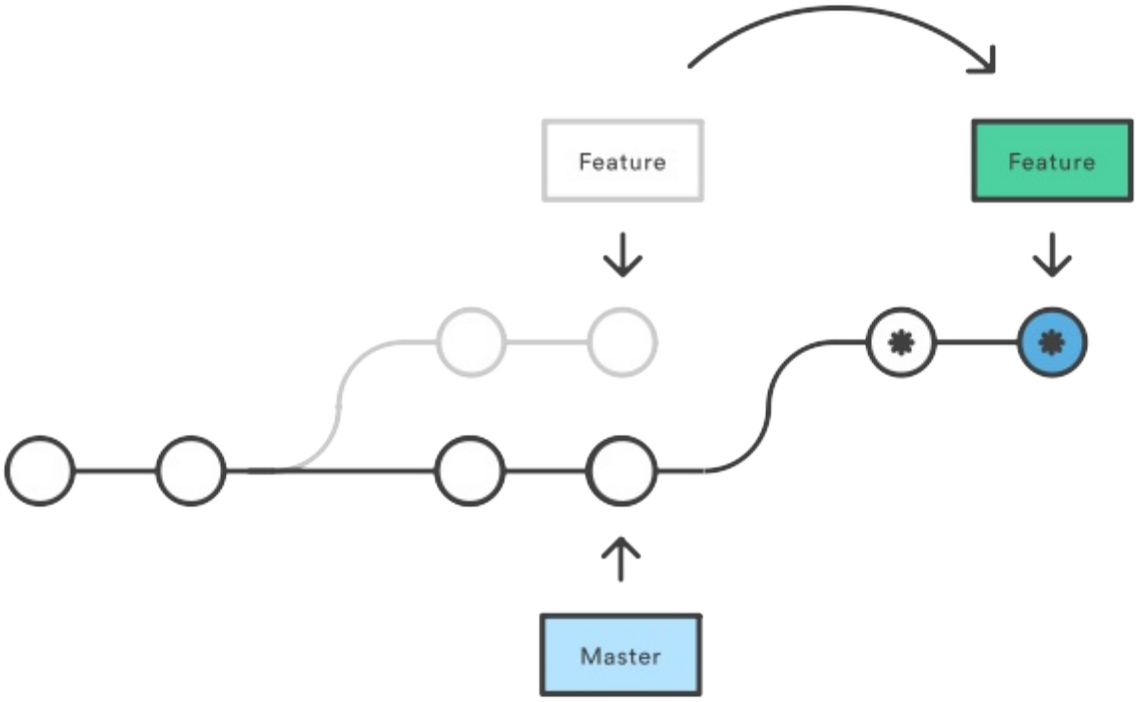
applying them to the specified base—it's literally rewriting your project history. It's very important to understand that, even though the branch looks the same, it's composed of entirely new commits.

The primary reason for rebasing is to maintain a linear project history. For example, consider a situation where the master branch has progressed since you started working on a feature as show below

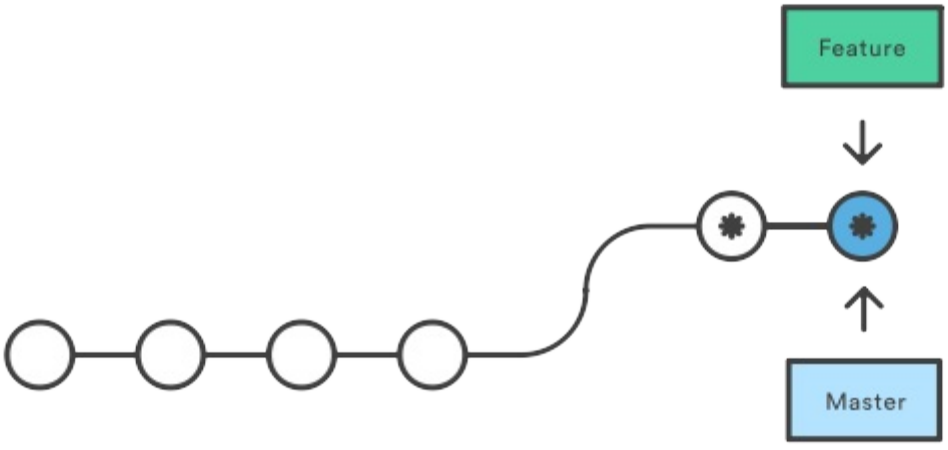


You have two options for integrating your feature into the master branch: merging directly or rebasing and then merging. The former option results in a 3-way merge and a merge commit that we have seen so far, while the latter results in a fast-forward merge and a perfectly linear history. The following diagram demonstrates how rebasing onto master facilitates a fast-forward merge.

After Rebasing Onto Master



After a Fast-Forward Merge



Brand New Commits

Rebasing is a common way to integrate upstream changes into your local repository. Pulling in upstream changes with git merge results in a superfluous merge commit every time you want to see how the project has progressed. On the other hand, rebasing is like saying, “I want to base my changes on what everybody has already done.”

As we've with git commit --amend and git reset, you should never rebase commits that have been pushed to a public repository. The rebase would replace the old commits with new ones, and it would look like that part of your project history abruptly vanished.

GitHub

GitHub is the single largest host for Git repositories, and is the central point of collaboration for millions of developers and projects. A large percentage of all Git repositories are hosted on GitHub, and many open-source projects use it for Git hosting, issue tracking, code review, and other things.

This course recommends that you use GitHub for your repositories. Teaching Assistants will help you set gitub account if you do not have one.

One constant headache working on a CPUT network when working with Git is the dreaded Proxy.

Below is the way to work around it

First, go to the root directory by typing the command below

```
$ cd
```

And then open a file called .gitconf

```
$ vim .gitconfig
```

Enter the following

```
[user]
        name = yourname
        email = ehatever@put.ac.za
[core]
        autocrlf = input
[http]
      proxy = http://username:password@155.238.4.87:8080

[https]
     proxy = http://username:password@155.238.4.87:8080
```

That should allow you to gro round the firewall issue and happily connect to GitHub

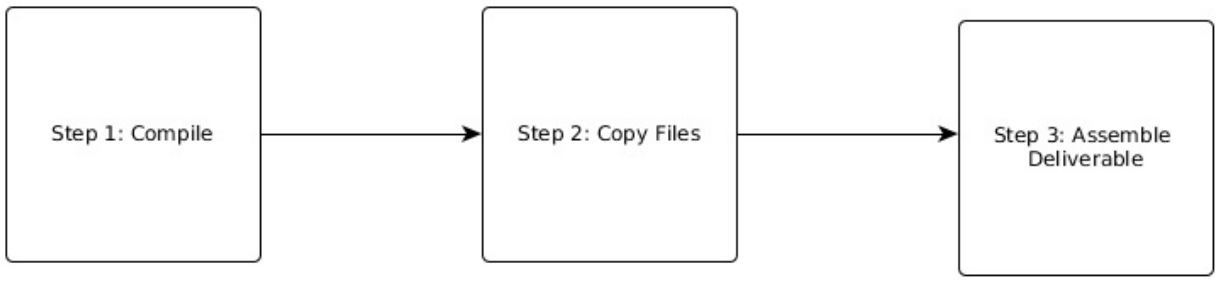
Build Tools

Efficient project automation is one of the key drivers in terms of delivering software to the end user. A good build tool should provide a developer with a flexible and maintainable way to model automation needs.

What is a Build Tool

A build too is programming utility that lets you express your automation needs as executable, ordered tasks.

For example, if you want to compile your source code, copy the generated class files into a directory, and assemble a deliverable that contains the class files. A deliverable could be a ZIP file, that can be distributed to a runtime environment, you will need this process automated .



Each of these tasks above represents a unit of work—for example, compilation of source code. The order is important. You can't create the ZIP archive if the required class files. Internally, tasks and their interdependencies are modeled as a directed acyclic graph (DAG).

Build tool should allow you to create a repeatable, reliable, and portable build without manual intervention.

The Anatomy of a Build Tool

A build tool has the following parts at the core: A Build File, Build Inputs and Outputs, Build Engine and a Depedency Manager.

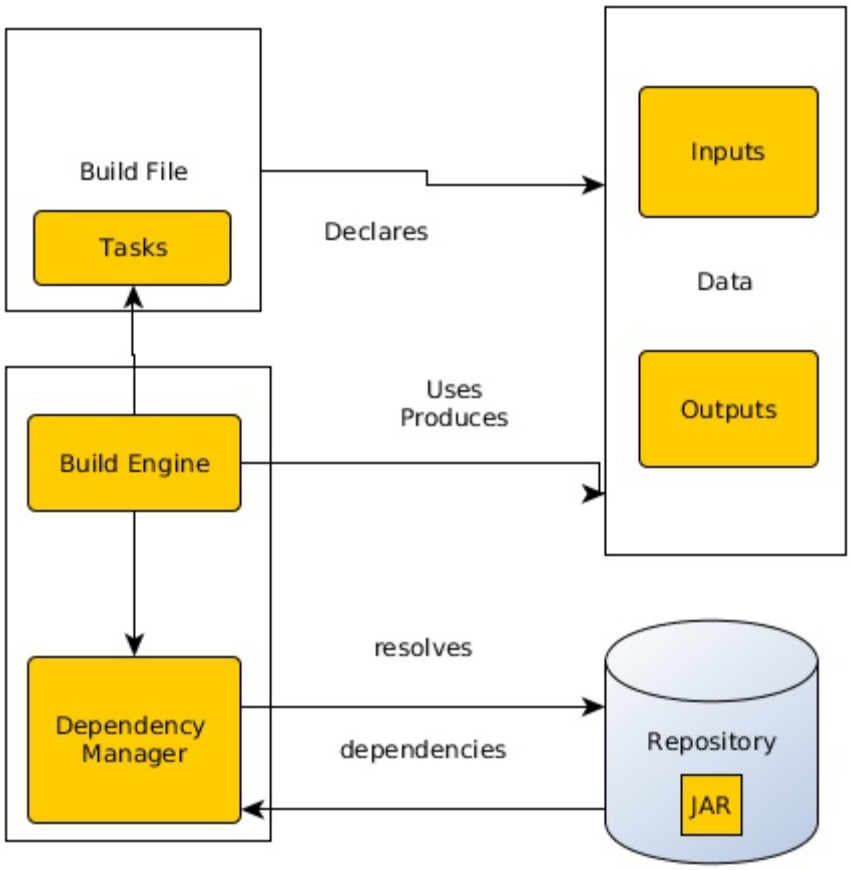
Build file : Contains the configuration needed for the build, defines external dependencies such as third-party libraries, and contains the instructions to achieve a specific goal in the form of tasks and their interdependencies.

Build Inputs and Outputs : A task takes an input, works on it by executing a series of steps, and produces an output. Some tasks may not need any input to function correctly, nor is creating an output considered mandatory. Complex task dependency graphs may use the output of a dependent task as input.

Build Engine: The build file's step-by-step instructions or rule set must be translated into an internal model the build tool can understand. The build engine processes the build file at runtime, resolves dependencies between tasks, and sets up the entire configuration needed to command the execution. Once the internal model is built, the engine will execute the series of tasks in the correct order.

Dependency Manager:The dependency manager is used to process declarative dependency definitions for the build file, resolve them from an artifact repository (for example, the local file system, an FTP , or an HTTP server), and make them available to your project.A dependency is generally an external, reusable library in the form of a JAR file. The repository acts as storage for dependencies, and organizes and describes them by identifiers, such as name and version. A typical repository can be an HTTP server or the local file system.

Diagram below summarise the anatomy of a build tool



Why We need Build Tools

Today it is vital to be able to build and deliver software in a repeatable and consistent way. Having to manually perform steps to produce and deliver software is time-consuming and error-prone.

Not only can you make mistakes along the way, manual intervention also takes away from the time you desperately need to get your actual work done. Any step in your software development process that can be automated should be automated.

Further, the actual building of your software usually follows predefined and ordered steps.For example, you compile your source code first, then run your tests, and lastly assemble a deliverable. You'll need to run the same steps over and over again—every day.The outcome of this process needs to be repeatable for everyone who runs the build.

Build tools also software build portable. An automated build shouldn't require a specific runtime environment to work, whether this is an operating system or an IDE . Optimally, the automated tasks should be executable from the command line, which allows you to run the build from any machine you want, whenever you want.

Java Build Tools

There areany build tools in the java space. These include Ant, Ivy, Gradle, SBT, Grape, Buildr, Leinigen and many more not listed here.

You will get a chance to use Gradle next semester when building Android Application For this term, we shall focus on the stable and widely used tool, though out of fashion , called Maven.

Apache Maven Build Tool

Apache Maven is an open source, standards-based project management framework that simplifies the building, testing, reporting, and packaging of projects.

Maven is based on the concept of a build lifecycle. Every project knows exactly which steps to perform to build, package, and distribute an application, including the fol- lowing functionality:

- Compiling source code
- Running unit and integration tests
- Assembling the artifact (for example, a JAR file)
- Deploying the artifact to a local repository
- Releasing the artifact to a remote repository

Life Cycle and Phases

Build processes generating artifacts typically require several steps and tasks to be completed successfully. Examples of such tasks include compiling source code, running a unit test, and packaging of artifacts. Maven uses the concept of goals to represent such granular tasks.

Every build follows a specified life cycle. Maven comes with a default life cycle that includes the most common build phases like compiling, testing and packaging.

The following lists gives an overview of the important Maven life cycle phases.

- validate - checks if the project is correct and all information is available
- compile - compiles source code in binary artifacts
- test - executes the tests
- package - takes the compiled code and package it, for example into a JAR file.
- integration-test - takes the packaged result and executes additional tests, which require the packaging
- verify - performs checks if the package is valid
- install - install the result of the package phase into the local Maven repository
- deploy - deploys the package to a target, i.e. remote repository

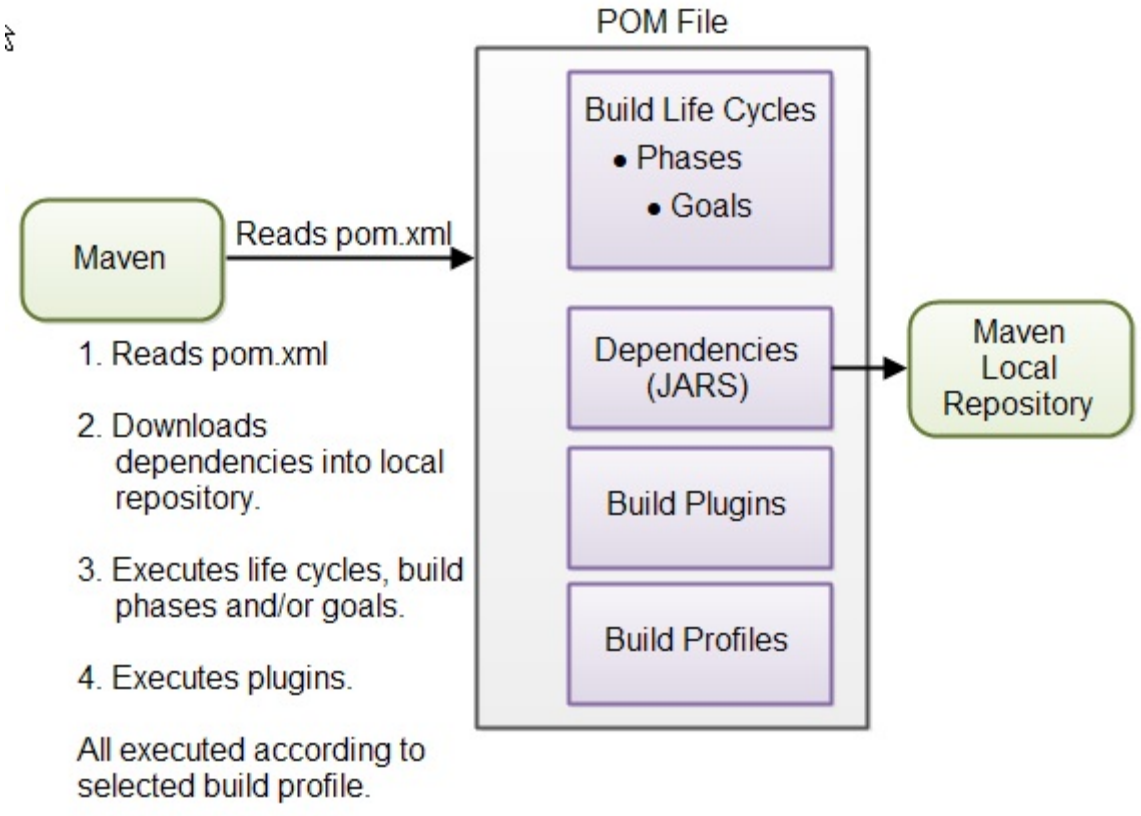
All these commands can be executed using

```
$ mvn [goal]
```

Maven Concepts

Maven is centered around the concept of POM files (Project Object Model). A POM file is an XML representation of project resources like source code, test code, dependencies (external JARs used) etc. The POM contains references to all of these resources. The POM file should be located in the root directory of the project it belongs to.

Here is a diagram illustrating how Maven uses the POM file, and what the POM file primarily contains:



POM Files

When you execute a Maven command you give Maven a POM file to execute the commands on. Maven will then execute the command on the resources described in the POM.

Build Life Cycles, Phases and Goals

The build process in Maven is split up into build life cycles, phases and goals. A build life cycle consists of a sequence of build phases, and each build phase consists of a sequence of goals. When you run Maven you pass a command to Maven. This command is the name of a build life cycle, phase or goal. If a life cycle is requested executed, all build phases in that life cycle are executed. If a build phase is requested executed, all build phases before it in the pre-defined sequence of build phases are executed too.

Dependencies and Repositories

One of the first goals Maven executes is to check the dependencies needed by your project. Dependencies are external JAR files (Java libraries) that your project uses. If the dependencies are not found in the local Maven repository, Maven downloads them from a central Maven repository and puts them in your local repository. The local repository is just a directory on your computer's hard disk. You can specify where the local repository should be located if you want to (I do). You can also specify which remote repository to use for downloading dependencies. All this will be explained in more detail later in this tutorial.

Build Plugins

Build plugins are used to insert extra goals into a build phase. If you need to perform a set of actions for your project which are not covered by the standard Maven build phases and goals, you can add a plugin to the POM file. Maven has some standard plugins you can use, and you can also implement your own in Java if you need to.

Build Profiles

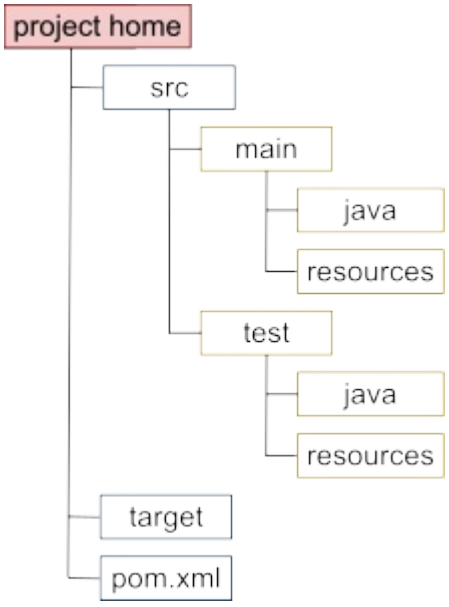
Build profiles are used if you need to build your project in different ways. For instance, you may need to build your project for your local computer, for development and test. And you may need to build it for deployment on your production environment. These two builds may be different. To enable different builds you can add different build profiles to your POM files. When executing Maven you can tell which build profile to use.

Maven POM Files

A Maven POM file (Project Object Model) is an XML file that describe the resources of the project. This includes the directories where the source code, test source etc. is located in, what external dependencies (JAR files) your projects has etc.

The POM file describes what to build, but most often not how to build it. How to build it is up to the Maven build phases and goals. You can insert custom actions (goals) into the Maven build phase if you need to, though.

Each project has a POM file. The POM file is named **pom.xml** and should be located in the root directory of your project as shown below .



A project divided into subprojects will typically have one POM file for the parent project, and one POM file for each subproject. This structure allows both the total project to be built in one step, or any of the subprojects to be built separately.

Here is a minimal POM file:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
   http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.kabaso.crawler</groupId>
    <artifactId>java-web-crawler</artifactId>
    <packaging>jar</packaging>
    <version>1.0.0</version>

        <dependencies>
            <dependency>
                <groupId>org.jsoup</groupId>
                <artifactId>jsoup</artifactId>
                <version>1.7.1</version>
            </dependency>
```

```
            <dependency>
                <groupId>junit</groupId>
                <artifactId>junit</artifactId>
                <version>4.8.1</version>
                <scope>test</scope>
            </dependency>

        </dependencies>

        <repositories>
            <repository>
                <id>spring.code</id>
                <url>http://maven.spring.com/maven2/lib</url>
            </repository>
        </repositories>

<build>
        <finalName>HelloApp</finalName>
        <plugins>
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>1.6</source>
                    <target>1.6</target>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>2.1</version>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>shade</goal>
                        </goals>
                        <configuration>
                            <transformers>
                                <transformer
                                        implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer
                                    <mainClass>com.kabaso.HelloApp</mainClass>
                                </transformer>
                            </transformers>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <artifactId>maven-surefire-plugin</artifactId>
                <configuration>
                    <includes>
                        <include>**/*Tests.java</include>
                    </includes>
                </configuration>
            </plugin>
        </plugins>
    </build>

     <profiles>
      <profile>
          <id>test</id>
          <activation>...</activation>
          <build>...</build>
          <modules>...</modules>
          <repositories>...</repositories>
          <pluginRepositories>...</pluginRepositories>
          <dependencies>...</dependencies>
          <reporting>...</reporting>
          <dependencyManagement>...</dependencyManagement>
          <distributionManagement>...</distributionManagement>
      </profile>
  </profiles>
</project>
```

Maven Archetypes

An archetype is a template for a Maven project which is used by the Maven Archetype plugin to create new projects.

You can use an archetype by invoking the generate goal of the Archetype plugin via the command-line as shown below

```
mvn archetype:generate \
-DgroupId=org.sonatype.mavenbook \
-DartifactId=quickstart \
-Dversion=1.0-SNAPSHOT \
-DpackageName=org.sonatype.mavenbook \
-DarchetypeGroupId=org.apache.maven.archetypes \
-DarchetypeArtifactId=maven-archetype-quickstart \
-DarchetypeVersion=1.0 \
-DinteractiveMode=false
```

The generate goal accepts the following parameters:

- **groupId** The groupId for the project you are creating.
- **artifactId** The artifactId for the project you are creating.
- **version** The version for the project you are creating (defaults to 1.0-SNAPSHOT).
- **packageName** The default package for the project you are creating (defaults to groupId).
- **archetypeGroupId** The groupId of the archetype you wish to use for project generation.
- **archetypeArtifactId** The artifactId of the archetype you wish to use for project generation.
- **archetypeVersion** The version of the archetype you wish to use for project generation.
- **interactiveMode** When the generate goal is executed in interactive mode, it will prompt the user for all the previously listed parameters. When interactiveMode is false, the generate goal will use the values passed in from the command line.

Chapter Exercises

Git Questions

These Execises Should all be done during th lab session in class. There will be lab assistant to help you with connection issues. All material you need to accomplish these exercises are in this chapter

1. Set up a GitHub account and address. Make sure that you choose a meaningful name that you will be able to easily recall and . For Example my GitHub account and address is http://github.com/boniface

2. Log into the Linux Image VM Player and create your unique folder and run the git commands for the following tasks and record he screen short of the results in a word document that you need to print and give to the lab assistants

    i. Initilaise git repository
    ii. Create a file in your repository and check the status and give the explanation of the status
    iii. Track he file that you just created and check status
    iv. Committ the file to your repository , check the status and give an explanation
    v. Create three more files and check the status and explain why the files e in the states they are in
    vi. Make Changes to the file you committed and check the status
    vii. Now track the two files that you added above and give the explanation of your status output
    viii. Commit the modified file ad check the status
    ix. Get the output of the Short Status of your repo
    x. Ensure that Git ignores one of the three files that you have created above
    xi. get the status of the changes that you made to your repository
    xii. Remove one of the three files that you had created and are tracking. Get the status and give an explanation
    xiii. Get the output of your committ History
3. This section requires your to use your GitHub account that you just created

    i. In your GitHub account, creaete a respository and clone it onto you PC and ensure that you put it in a different directory 2.show your full remotes on your local machine using a git command and take a snapshort
    ii. Add a file to your repo and push he file to GitHub
    iii. Create Another Git Repo and add any other reomote repository and again show full remotes for your repository
    iv. Get the contents of the remote repository in and merge it into your local repo
    v. Create another git repository and add another repo repository.
    vi. Get the content using using one git command to so that the result is a fully merged repo
    vii. Remove the remote from your repostory
4. Demostrate the ability to tag your repo with version numbers and commit these them to gitHub

5. Read the chapter on branching and mostrate the ability to create branches and integrating the merge. GitHub will produce graphs for you.

6. Demostrate Rebasing on another repo that you create on GitHub

Maven Questions

Chapter 2: Testing, Test Driven Development

Chapter Objectives

1. Explain and demonstrate Unit and integration testing
2. Use the important technique called Test-Driven Development, which enables de- velopers to write the tests that prove their code is correct before they write their code.
3. Learn tools, and patterns for using testing to deliver good, defect free code as fast as possible.

Testing

Introduction To Testing

There are lots of different kinds of testing that can and should be performed on a software project. Some of this testing requires extensive involvement from the end users; other forms may require teams of dedicated Quality Assurance personnel or other expensive resources.

But that's not what we're going to talk about in this module. Instead, we're talking about unit testing: an essential, if often misunderstood, part of project and personal success.

Unit testing is a relatively inexpensive, easy way to produce better code, faster. A unit test is a piece of code written by a developer that exercises a very small, specific functionality of the code being tested. Usually a unit test exercises some particular method in a particular context. Unit testing will make your life easier.

It will make your designs better and drastically reduce the amount of time you spend debugging. When writing test code, there are some naming conventions you need to follow. If you have a method named **createAccount** that you want to test, then your first test method might be named **testCreateAccount**. The method **testCreateAccount** will call cre- ateAccount with the necessary parameters and verify that **createAccount** works as advertised.You can, of course, have many test methods that exercise **createAccount**. If you have a class called **Calculator** that you need to unit test, you would call the class **CalculatorTest**

The test code must be written to do a few things:

1. Setup all conditions needed for testing (create any required objects, allocate any needed resources, etc.)
2. Call the method to be tested
3. Verify that the method to be tested functioned as expected
4. Clean up after itself

You write test code and compile it in the normal fashion, as you would any other bit of source code in your project.

As we've seen, there are some helper methods that assist you in determining whether a method under test is performing

There are some helper methods that assist you in determining whether a method under test is performing correctly or not. Generically, we call all these methods asserts. They let you assert that some condition is true; that two bits of data are equal, or not, and so on. We'll take a look at each one of the assert methods that JUnit provides next.

Important Unit Testing Methods

In this course we shall be using JUnit 4 as a Testing Framework. JUnit 4 uses Java 5 annotations to completely eliminate both of these conventions. The class hierarchy is no longer required and methods intended to function as tests need only be decorated with a newly defined **@Test** annotation. Java 5 annotations make JUnit 4 a notably different framework from previous versions. Declaring a test in JUnit 4 is a matter of decorating a test method with the **@Test** annotation.

Fixtures

Sometimes the setup can be more complex than you want to put in a constructor or a field initializer Sometimes you want to reinitialize static data Sometimes you just want to share setup code between different methods.

Fixtures foster reuse through a contract that ensures that particular logic is run either before or after a test. In older versions of JUnit, this contract was implicit regardless of whether you implemented a fixture or not. JUnit 4, however, has made fixtures explicit through annotations, which means the contract is only enforced if you actually decide to use a fixture.

Through a contract that ensures fixtures can be run either before or after a test, you can code reusable logic.

This logic, for example, could be initializing a class that you will test in multiple test cases or even logic to populate a database before you run a data-dependent test. Either way, using fixtures ensures a more manageable test case: one that relies on common logic.

Fixtures come in especially handy when you are running many tests that use the same logic and some or all of them fail. Rather than sifting through each test's set-up logic, you can look in one place to deduce the cause of failure.

In addition, if some tests passand others fail, you might be able to avoid examining the fixture logic as a source of the failures altogether.

JUnit 4 uses annotations to cut out a lot of the overhead of fixtures, allowing you to run a fixture for every test or just once for an entire class or not at all. There are four fixture annotations: two for class-level fixtures and two for method-level ones.

At the class level, you have **@BeforeClass** and **@AfterClass**, and at the method (or test) level, you have **@Before** and **@After**. :0

```
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Ignore;
import org.junit.Test;

public class TestMethods {

   //execute before class
   @BeforeClass
   public static void beforeClass() {
      System.out.println("in before class");
   }

   //execute after class
   @AfterClass
   public static void  afterClass() {
      System.out.println("in after class");
   }

   //execute before test
   @Before
   public void before() {
      System.out.println("in before");
   }

   //execute after test
   @After
   public void after() {
      System.out.println("in after");
   }

   //test case
   @Test
   public void test() {
      System.out.println("in test");
   }

   //test case ignore and will not execute
   @Ignore
   public void ignoreTest() {
      System.out.println("in ignore test");
   }
```

```
    //Fails if the method does not throw the named exception.
   @Test (expected = Exception.class)
   public void testException() {

   }

    //Fails if the method takes longer than 100 milliseconds.
    @Test(timeout=100)
   public void testTimeOut() {

   }

}
```

Assert statements

JUnit provides static methods in the Assert class to test for certain conditions. These assertion methods typically start with assert and allow you to specify the error message, the expected and the actual result. An assertion method compares the actual value returned by a test to the expected value, and throws an AssertionException if the comparison test fails.

The following table gives an overview of these methods. Parameters in [] brackets are optional.

Statement	Description
fail(String)	Let the method fail. Might be used to check that a certain part of the code is not reached or to have a failing test before the test code is implemented. The String parameter is optional.
assertTrue([message], boolean condition)	Checks that the boolean condition is true.
assertFalse([message], boolean condition)	Checks that the boolean condition is false.
assertEquals([String message], expected, actual)	Tests that two values are the same. Note: for arrays the reference is checked not the content of the arrays.
assertEquals([String message], expected, actual, tolerance)	Test that float or double values match. The tolerance is the number of decimals which must be the same.
assertNull([message], object)	Checks that the object is null.
assertNotNull([message], object)	Checks that the object is not null.
assertSame([String], expected, actual)	Checks that both variables refer to the same object.
assertNotSame([String], expected, actual)	Checks that both variables refer to different objects.

Test execution order

JUnit assumes that all test methods can be executed in an arbitrary order. Well-written test code should not assume any order, i.e., tests should not depend on other tests.

As of JUnit 4.11 you can use an annotation to define that the test methods are sorted by method name, in lexicographic order.

To activate this feature, annotate your test class with

```
@FixMethodOrder(MethodSorters.NAME_ASCENDING).
```

Test Driven Development

Introduction

Test-driven development (TDD) is an advanced technique of using automated unit tests to drive the design of software and force decoupling of dependencies. The result of using this practice is a comprehensive suite of unit tests that can be run at any time to provide feedback that the software is still working. The motto of Test-Driven Development is RED, GREEN, REFACTOR.

- RED : Create a test and make it fail.
- GREEN: Make the test pass by any means necessary.
- REFACTOR: Change the code to remove duplication in your project and to improve the design while ensuring that all tests still pass.

The /Red/Green/Refactor cycle is repeated very quickly for each new unit of code.

Process Example

Follow these steps :

1. Understand the requirements of the story, work item, or feature that you are working on.
2. RED: Create a test and make it fail.
    - Imagine how the new code should be called and write the test as if the code already existed. You will not get IntelliSense because the new method does not yet exist.
    - Create the new production code stub. Write just enough code so that it compiles.
    - Run the test. It should fail. This is a calibration measure to ensure that your test is calling the correct code and that the code is not working by accident. This is a meaningful failure, and you expect it to fail.
3. GREEN: Make the test pass by any means necessary.

    - Write the production code to make the test pass. Keep it simple.
    - Some advocate the hard-coding of the expected return value first to verify that the test correctly detects success. This varies from practitioner to prac- titioner.
    - If you've written the code so that the test passes as intended, you are finished. You do not have to write more code speculatively. The test is the objective definition of "done." The phrase "You Ain't Gonna Need It" (YAGNI) is often used to veto unnecessary work. If new functionality is still needed, then another test is needed. Make this one test pass and continue.
    - When the test passes, you might want to run all tests up to this point to build confidence that everything else is still working.
4. REFACTOR: Change the code to remove duplication in your project and to im- prove the design while ensuring that all tests still pass.

    - Remove duplication caused by the addition of the new functionality.
    - Make design changes to improve the overall solution.
    - After each refactoring, rerun all the tests to ensure that they all still pass.
5. Repeat the cycle. Each cycle should be very short, and a typical hour should contain many RED/GREEN/REFACTOR cycles.

Benefits of Test-Driven Development

1. The suite of unit tests provides constant feedback that each component is still working.

2. The unit tests act as documentation that cannot go out-of-date, unlike separate documentation, which can and frequently does.
3. When the test passes and the production code is refactored to remove duplication, it is clear that the code is finished, and the developer can move on to a new test.
4. Test-driven development forces critical analysis and design because the developer cannot create the production code without truly understanding what the desired result should be and how to test it.
5. The software tends to be better designed, that is, loosely coupled and easily main- tainable, because the developer is free to make design decisions and refactor at any time with confidence that the software is still working. This confidence is gained by running the tests. The need for a design pattern may emerge, and the code can be changed at that time.
6. The test suite acts as a regression safety net on bugs: If a bug is found, the developer should create a test to reveal the bug and then modify the production code so that the bug goes away and all other tests still pass. On each successive test run, all previous bug fixes are verified.
7. Reduced debugging time!

Characteristics of a Good Unit Test

A good unit test has the following characteristics.

1. Runs fast, runs fast, runs fast. If the tests are slow, they will not be run often.
2. Separates or simulates environmental dependencies such as databases, file systems, networks, queues, and so on. Tests that exercise these will not run fast, and a failure does not give meaningful feedback about what the problem actually is.
3. Is very limited in scope. If the test fails, it's obvious where to look for the problem. Use few Assert calls so that the offending code is obvious. It's important to only test one thing in a single test.

4. Runs and passes in isolation. If the tests require special environmental setup or fail unexpectedly, then they are not good unit tests. Change them for simplicity and reliability. Tests should run and pass on any machine. The "works on my box" excuse doesn't work.

5. Often uses stubs and mock objects. If the code being tested typically calls out to a database or file system, these dependencies must be simulated, or mocked. These dependencies will ordinarily be abstracted away by using interfaces.
6. Clearly reveals its intention. Another developer can look at the test and under- stand what is expected of the production code.

Chapter Exercises

1. By Using Maven as a build tool, Write Small programes that demonstrate test fixture for the following
    i. Floating Point
    ii. Integers
    iii. Object Equality
    iv. Object Identity
    v. Truth
    vi. False
    vii. Nullness
    viii. Non Nullness
    ix. Failing Test
    x. Exceptions 11.Timeouts
    xi. DisablingTest
    xii. Arrays Content
2. Create a Multi-Module project using Maven. Using TDD show that the the modules work together to deliver the functionality of the application you have come up with. Your Application should have at least four modules independtely testable.

Chapter 3: Java Data Structures Concepts and Coding to an Interface

Chapter Objectives

1. Differentiate the terms Abstract Class, Concrete Class and Interface
2. Know the difference between overriding and Overloading
3. Learn the Java Generic class. Focus in Collections, Lists, Sets, Map
4. Learn how to code to an Interface

Introduction

This section introduces some basic data structures. A Datastructure is a set of types, a set of functions, and a set of axioms, implying that a datastructure is a type with implementation. In our object-oriented programming, type with implementation means concrete class.

The datastructure is a class definition is too broad because it embraces Employee, Vehicle, Account, and many other real-world entity-specific classes as datastructures. Although those classes structure various data items, they do so to describe real-world entities (in the form of objects) instead of describing container objects for other entity (and possibly container) objects.

This containment idea leads to a more appropriate datastructure definition: a datastructure is a container class that provides storage for data items, and capabilities for storing and retrieving data items. Examples of container datastructures: arrays, linked lists, stacks, and queues.

In this course, we will focus on a few inbuilt data structures like Collection, List, Set and Map. These are datastructures are are crucial for navigating through this course.

First of all, we will start with the recap of Java's OO concepts

Interface , Abstract, Concrete Classes and Annotations

Interface

A Java interface is a class, which only declare abstract methods and variables. An abstract methods is a method without an mplementation body. Interfaces are a way to achieve polymorphism in Java.

Below is an example Interface

```
 public interface MyInterface {

    public String hello = "Hello World";

    public void sayHello();
}
```

An interface is declared using the Java keyword interface. Just like with classes, an interface can be declared public or package scope (no access modifier).

The interface contains one variable and one method. The variable can be accessed directly from the interface, like this:

```
MyInterface.hello;
```

The method, however, needs to be implemented by some class, before you can access it as show below

```
public class MyInterfaceImpl implements MyInterface {

    public void sayHello() {
        System.out.println(MyInterface.hello);
    }
}
```

And then the usage

```

MyInterface myInterface = new MyInterfaceImpl();

myInterface.sayHello();
```

Note that A class can implement multiple interfaces. In that case the class must implement all the methods declared in all the interfaces implemented

Abstract

Abstract classes in Java are classes, like Interfaces, which cannot be instantiated. The purpose of an abstract class is to function as a base for subclasses.

You declare that a class is abstract by adding the abstract keyword to the class declaration.

```
public abstract class MyAbstractClass {

}
```

The difference between an Interface and an abstract class is that while an interface has only abstract methods, and abstract class can have concrete methods.

An abstract class can have abstract methods. You declare a method abstract by adding the abstract keyword in front of the method declaration.

```
public abstract class MyAbstractClass {

    public abstract void abstractMethod();
}
```

An abstract method has no implementation. It just has a method signature.

Below is a sample abstract class with two concrete methods and one abstract method

```
public abstract class MyAbstractProcess {

    public void process() {
        stepBefore();
        action();
        stepAfter();
    }

    public void stepBefore() {
        //implementation directly in abstract superclass
    }

    public abstract void action(); // implemented by subclasses

    public void stepAfter() {
        //implementation directly in abstract superclass
    }
}
```

Concrete Class

A Java concrete class is a template for how objects of that class looks. A concrete class is used to create Java objects.

To create objects of a certain class, you use the new keyword as shown in the diagram below where three car objects are created from the concrete Car class

```
public class Car {

}

Car car1 = new Car();
Car car2 = new Car();
Car car3 = new Car();
```

Annotations

Java annotations are used to provide meta data for to Java code. Being meta data, the annotations do not directly

affect the execution of your code, although some types of annotations can actually be used for that purpose.

Java annotations were added to Java from Java 5.

Java annotations are typically used for the following purposes:

1. Compiler instructions
2. Build-time instructions
3. Runtime instructions

An annotation in its shortest form looks like this:

```
@Test
```

The @ character signals to the compiler that this is an annotation. The name following the @ character is the name of the annotation. In the example above the annotation name is Test

You can put Java annotations above classes, interfaces, methods, method parameters, fields and local variables. Here is an example with anotations

```
@Entity
public class Vehicle {

    @Persistent
    protected String vehicleName = null;


    @Getter
    public String getVehicleName() {
        return this.vehicleName;
    }

    public void setVehicleName(@Optional vehicleName) {
        this.vehicleName = vehicleName;
    }

    public List addVehicleNameToList(List names) {

        @Optional
        List localNames = names;

        if(localNames == null) {
            localNames = new ArrayList();
        }
        localNames.add(getVehicleName());

        return localNames;
    }

}
```

Java comes with three annotations which are used to give the Java compiler instructions. These annotations are:

- @Deprecated
- @Override
- @SuppressWarnings

Collections, List, Set and Map

Introduction

The Java Collections API's provide Java developers with a set of classes and interfaces that makes it easier to handle collections of objects.

Rather than having to write your own collection classes, Java provides these ready-to-use collection classes for you.

The purpose of this tutorial is to give you an overview of the Java Collection classes. Thus it will not describe each and every little detail of the Java Collection classes. But, once you have an overview of what is there, it is much easier to read the rest in the JavaDoc's afterwards.

Most of the Java collections are located in the java.util package. In this course, we will just focus on Four collections API, namely Collection, List, Set and Map

Collection

The Collection interface (java.util.Collection) is one of the root interfaces of the Java collection classes. Though you do not instantiate a Collection directly, but rather a subtype of Collection, you may often treat these subtypes uniformly as a Collection

Java does not come with a usable implementation of the Collection interface, so you will have to use one of the subtypes(List, Set, Map and Tree.

The Collection interface just defines a set of methods (behaviour) that each of these Collection subtypes share. This makes it possible ignore what specific type of Collection you are using, and just treat it as a Collection.

This is standard inheritance, so there is nothing magical about, but it can still be a nice feature from time to time. Later sections in this text will describe the most used of these common operations.

List

The java.util.List interface is a subtype of the java.util.Collection interface. It represents an ordered list of objects, meaning you can access the elements of a List in a specific order, and by an index too. You can also add the same element more than once to a List.

Being a Collection subtype all methods in the Collection interface are also available in the List interface.

Since List is an interface you need to instantiate a concrete implementation of the interface in order to use it. You can choose between the following List implementations in the Java Collections API:

- java.util.ArrayList
- java.util.LinkedList
- java.util.Vector
- java.util.Stack

Below is the instatiation of a List

```
List listA = new ArrayList();
List listB = new LinkedList();
```

```
List listC = new Vector();
List listD = new Stack();
```

To add elements to a List you call its add() method. This method is inherited from the Collection interface. Here are a few examples:

```
List listA = new ArrayList();

listA.add("element 1");
listA.add("element 2");
listA.add("element 3");

listA.add(0, "element 0");
```

The first three add() calls add a String instance to the end of the list. The last add() call adds a String at index 0, meaning at the beginning of the list.

The order in which the elements are added to the List is stored, so you can access the elements in the same order.

You can remove elements in two ways:

- remove(Object element)
- remove(int index)

remove(Object element) removes that element in the list, if it is present. All subsequent elements in the list are then moved up in the list. Their index thus decreases by 1.

remove(int index) removes the element at the given index. All subsequent elements in the list are then moved up in the list. Their index thus decreases by 1.

Set

The java.util.Set interface is a subtype of the java.util.Collection interface. It represents set of objects, meaning each element can only exists once in a Set.

Being a Collection subtype all methods in the Collection interface are also available in the Set interface.

Since Set is an interface you need to instantiate a concrete implementation of the interface in order to use it. You can choose between the following Set implementations in the Java Collections API:

- java.util.EnumSet
- java.util.HashSet
- java.util.LinkedHashSet
- java.util.TreeSet

Here are a few examples of how to create a Set instance:

```
Set setA = new EnumSet();
Set setB = new HashSet();
Set setC = new LinkedHashSet();
Set setD = new TreeSet();
```

To add elements to a Set you call its add() method. This method is inherited from the Collection interface. Here are a few examples:

```
Set setA = new HashSet();

setA.add("element 1");
setA.add("element 2");
setA.add("element 3");
```

You remove elements by calling the remove(Object o) method. There is no way to remove an object based on index in a Set, since the order of the elements depends on the Set implementation.

Map

The java.util.Map interface represents a mapping between a key and a value. The Map interface is not a subtype of the Collection interface. Therefore it behaves a bit different from the rest of the collection types.

Since Map is an interface you need to instantiate a concrete implementation of the interface in order to use it. You can choose between the following Map implementations in the Java Collections API:

- java.util.HashMap
- java.util.Hashtable
- java.util.EnumMap
- java.util.IdentityHashMap
- java.util.LinkedHashMap
- java.util.Properties
- java.util.TreeMap
- java.util.WeakHashMap

Here are a few examples of how to create a Map instance:

```
Map mapA = new HashMap();
Map mapB = new TreeMap();
Map mapC = new Hashtable();
Map mapD = new EnumMap();
Map mapE = new IdentityHashMap();
Map mapF = new Properties();
Map mapF = new LinkedHashMap();
Map mapH = new WeakHashMap();
```

To add elements to a Map you call its put() method. Here are a few examples:

```
Map mapA = new HashMap();

mapA.put("key1", "element 1");
mapA.put("key2", "element 2");
mapA.put("key3", "element 3");
```

The three put() calls maps a string value to a string key. You can then obtain the value using the key. To do that you use the get() method like this:

```
String element1 = (String) mapA.get("key1");
```

You remove elements by calling the remove(Object key) method. You thus remove the (key, value) pair matching the key.

Generic Collections

It is possible to specify the various Collection and Map types and subtypes in the Java collection API.

The Collection interface can be specified like this:

```
Collection<Dog> dogCollection = new HashSet<Dog>();
```

This dogCollection can now only contain Dog instances. If you try to add anything else, or cast the elements in the collection to any other type than Dog, the compiler will complain.

Coding to an Interface

To Demostrate how you code to an Interface, we will look at a simple program called Calculator that is coding to a concrete class. Here is the Simple Calculator concrete class

```
public class Calculator {

    int add(int a, int b) {
        return a+b;
    }
}
```

We can run this using a Test case below

```
// Other parts Omitted for clarity
    @Test
     public void add() {
     Calculator calc = new Calculator();
     int result = calc.add(10,20);
     Assert.assertEquals("Add 2+3", 30, result);
     }
```

This calc object depends on the concrete class of type Calculator and f we we make changes to the Class, i.e swapping out a different implementation, we will have to make changes to the calc object as well. So this implementation is rigid and not flexible to allow the client and the calculator to evolve independently

Let us introduce the interface o that we seperate e implementation from the contract

```
public interface CalculatorInterface {
    int add(int a, int b);
}
```

With the Interface Done we can now create the implementation of this contract

```
public class CalculatorServiceImpl implements CalculatorInterface{
    @Override
    public int add(int a, int b) {
        return a+b;
    }
}
```

With the Implemenation done, we can now update our test method to depend on an Interface type

```
// Other parts Omitted for clarity
    @Test
     public void add() {
     CalculatorInterface calc = new CalculatorServiceImpl();
     int result = calc.add(10,20);
     Assert.assertEquals("Add 2+3", 30, result);
     }
```

This is a much improved version over the first cut. It still has drawbacks because the implemtation is still leaking into the test client. To solve this problem, we need to introduce the Springframework IoC or Dependency injection Container.

The first thing we need to do is add the SpringLibrary to our MAven POM as shown below

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>calculator</groupId>
    <artifactId>calculator</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>calculator</name>
    <url>http://maven.apache.org</url>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>

    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
    </dependency>

 <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>4.1.4.RELEASE</version>
</dependency>

    </dependencies>
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-framework-bom</artifactId>
                    <version>4.1.4.RELEASE</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
</project>
```

Next up is we need to create a configration package where we will put files to help us wire our dependencies. It is in this file where we shall be wirig our dependencies

```
@Configuration
public class AppConfig {
    @Bean(name="calc")
    public CalculatorInterface getService(){
        return new CalculatorServiceImpl();
    }

}
```

With the Config in place, we can now update our test class to the code below

```
public class CalculatorTest {
    private CalculatorInterface calc;

    @BeforeMethod
    public void setUp() throws Exception {
        ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);
        calc = (CalculatorInterface)ctx.getBean("calc");
```

```
    }

    @AfterMethod
    public void tearDown() throws Exception {

    }
    @Test
    public void testAdd() throws Exception {
        int result = calc.add(5,5);
        Assert.assertEquals(result,10," Sum of two numbers 5 +5 is 10");
    }
```

As you can see above, our test client is now completely decoupled from the implementation class CalculatorServiceImpl. With this in place, we can now swap out the implementation without having to change the any code on the client side.

Chapter Exercises

1. Create an Application that will make use of Collection, List, Set and Map
2. Create an application that makes use of the coding to the Interface without the use of the Springramework
3. Refactor your application in (2) to add the Springframework and add multiple implemetation of your interfaces

Please not that all Exercises should use Test Driven Development Principles. For every Feature, there has to be a Test for it.

Chapter 4: Application Design Concepts and Principles

Chapter Objectives

1. Explain and demonstrate the main advantages of an object ori- ented approach to system design including the effect of encap- sulation, inheritance, delegation, and the use of interfaces, on architectural issues.
2. Describe how the principle of separation of concerns has been applied to the main system design
3. Describe and apply software design Concepts to system designs.

Fundamentals of Object Oriented Programming Concepts

There are three major features in object-oriented programming: encapsulation, inheritance and polymorphism.

Encapsulation

Encapsulation enforces modularity.

Encapsulation refers to the creation of self-contained modules that bind processing func- tions to the data. These user-defined data types are called classes, and one instance of a class is an object. In other words encapsulation means that the attributes (data) and the behaviors (code) are encapsulated in to a single object.

In other models, namely a structured model, code is in files that are separate from the data. An object, conceptually, combines the code and data into a single entity.

In programing at the class level, Encapsulation is the technique of making the fields in a class private and providing access to the fields via public methods. If a field is declared private, it cannot be accessed by anyone outside the class, thereby hiding the fields within the class.

For this reason, encapsulation is also referred to as data hiding. Encapsulation can be described as a protective barrier that prevents the code and data being randomly accessed by other code defined outside the class.

Access to the data and code is tightly controlled by an interface. The main benefit of encapsulation is the ability to modify our implemented code without breaking the code of others who use our code. With this feature Encapsulation gives maintainability, flexibility and extensibility to our code.

The code below shows a class which is encapsulated.

Benefits of Encapsulation include:

1. The fields of a class can be made read-only or write-only.
2. A class can have total control over what is stored in its fields.
3. The users of a class do not know how the class stores its data. A class can change the data type of a field, and users of the class do not need to change any of their code.

Inheritance

Inheritance passes knowledge down.

Inheritance can be defined as the process where one object acquires the properties of another. With the use of inheritance the information is made manageable in a hierarchical order. When we talk about inheritance the most commonly used keyword would be extends and implements. These words would determine whether one object **IS-A** type of another.

By using these keywords we can make one object acquire the properties of another object.

Usually classes are created in hierarchies, and inheritance allows the structure and methods in one class to be passed down the hierarchy. That means less programming is required when adding functions to complex systems.

If a step is added at the bottom of a hierarchy, then only the processing and data associated with that unique step needs to be added. Everything else about that step is inherited. The ability to reuse existing objects is considered a major advantage of object technology.

One of the major design issues in **O-O** programming is to factor out commonality of the various classes.

For example, say you have a **Dog class** and a **Cat class**, and each will have an attribute for eye color. In a procedural model, the code for Dog and Cat would each contain this attribute. In an **O-O design**, the color attribute can be abstracted up to a class called Mammal along with any other common attributes and methods.

Lets revisit Encapsulation: One of the primary advantages of using objects is that the object need not reveal all of its attributes and behaviors.

In good **O-O design** , an object should only reveal the interfaces needed to interact with it. Details not important to the use of the object should be hidden from other objects. This is called encapsulation. The interface is the fundamental means of communication between objects. Each class design specifies the interfaces for the proper instantiation and operation of objects. Any behavior that the object provides must be invoked by a message sent using one of the provided interfaces. The interface should completely describe how users of the class interact with the class.

In Java, the methods that are part of the interface are designated as public; everything else is part of the private implementation.

This is The Encapsulation Rule: Whenever the interface/implementation paradigm is covered, you are really talking about encapsulation. The basic question is what in a class should be exposed and what should not be exposed. This encapsulation pertains equally to data and behavior.

When talking about a class, the primary design decision revolves around encapsulating both the data and the behavior into a well-written class. In other words the process of packaging your program, dividing each of its classes into two distinct parts: the interface and the implementation is Encapsulation in the big picture.

Encapsulation is so crucial to O-O development that it is one of the generally accepted object-oriented designs cardinal rules.

Yet, Inheritance is also considered one of the four primary O-O concepts. However, in one way, inheritance actually breaks encapsulation!

How can this be? Is it possible that two of the three primary concepts of O-O are incompatible with each other?

There are three criteria that determine whether or not a language is object-oriented: encapsulation, inheritance, polymorphism.

To be considered a true object-oriented language, the language must support all three of these concepts.

Because encapsulation is the primary object-oriented mandate, you must make all at- tributes private. Making the attributes public is not an option.

Thus, it appears that the use of inheritance may be severely limited. If a subclass does not have access to the attributes of its parent, this situation presents a sticky design problem. To allow subclasses to access the attributes of the parent, language designers have included the access modifier protected. There are actually two types of protected access.

Rule of thumb is **AVOID CONCRETE INHERITANCE!!!** Only use it when it has been imposed on your and you

cannot change the class you are reusing. What is the Inheritance alternative? **Favour Composition over inheritance.**

Polymorphism

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object. Any java object that can pass more than one IS-A test is considered to be polymorphic. In Java, all java objects are polymorphic since any object will pass the IS-A test for their own type and for the class Object. It is important to know that the only possible way to access an object is through a reference variable.

A reference variable can be of only one type. Once declared the type of a reference variable cannot be changed. The reference variable can be reassigned to other objects provided that it is not declared final.

The type of the reference variable would determine the methods that it can invoke on the object. A reference variable can refer to any object of its declared type or any subtype of its declared type. A reference variable can be declared as a class or interface type.

Let us look at an example.

Now the Cow class is considered to be polymorphic since this has multiple inheritance. Following are true for the above example:

- Cow IS-A aAnimal
- Cow IS-A Vegetarian
- Cow IS-A Cow
- Cow IS-A Object

```
public interface Vegetarian {}
public class Animal {}
public class Cow extends Animal implements Vegetarian{}
```

```
Cow c = new Cow();
Animal a = c;
Vegetarian v = c;
Object o = c;
```

All the reference variables **c,a,v,o** refer to the same Cow object in the heap.

Design Principles

Great software is built in an agile way. Agility is about building software in tiny incre- ments, it is important to take the time to ensure that the software has a good structure that is flexible, maintainable, and reusable.

In an agile team, the big picture evolves along with the software. With each iteration, the team improves the design of the system so that it is as good as it can be for the system as it is now.

The team does not spend very much time looking ahead to future requirements and needs. Nor does it try to build in today the infrastructure to support the features that may be needed tomorrow.

Rather, the team focuses on the current structure of the system, making it as good as it can be. This is a way to incrementally evolve the most appropriate architecture and design for the system. It is also a way to keep that design and architecture appropriate as the system grows and evolves over time.

Agile development makes the process of design and architecture continuous. How do we know how whether the design of a software system is good? Below are the tips to help us detect bad design

The symptoms are:

1. **Rigidity:** The design is difficult to change.Rigidity is the tendency for software to be difficult to change. A design is rigid if a single change causes a cascade of subsequent changes in dependent modules.
2. **Fragility** is the tendency of a program to break in many places when a single change is made.
3. **Immobility:** A design is immobile when it contains parts that could be useful in other systems, but the effort and risk involved with separating those parts from the original system are too great.
4. **Viscosity.** It is difficult to do the right thing. Viscosity comes in two forms: viscosity of the software and viscosity of the environment. Goal is to design our software such that the changes that preserve the design are easy to make. Viscos- ity of environment comes about when the development environment is slow and inefficient In both cases, a viscous project is one in which the design of the software is difficult to preserve. We want to create systems and project environments that make it easy to preserve and improve the design.
5. **Needless complexity.** Overdesign. A design smells of needless complexity when it contains elements that aren't currently useful. This frequently happens when developers anticipate changes to the requirements and put facilities in the software to deal with those potential changes.
6. **Needless repetition.** Mouse abuse.Cut and paste may be useful text-editing operations, but they can be disastrous code-editing operations. Bad code can easily propagate to all modules and make it hard to change.
7. **Opacity.** Disorganized expression. Opacity is the tendency of a module to be difficult to understand. Code can be written in a clear and expressive manner, or it can be written in an opaque and convoluted manner. Code that evolves over time tends to become more and more opaque with age. A constant effort to keep the code clear and expressive is required in order to keep opacity to a minimum.

Software Design Principles

The Single-Responsibility Principle (SRP)

The Single Responsibility Principle (SRP) states that a class or module should have one, and only one, reason to change.

In the context of the SRP, we define a responsibility to be a reason for change. If you can think of more than one motive for changing a class, that class has more than one responsibility

If a class assumes more than one responsibility, that class will have more than one reason to change. Why SRP matters

1. We want it to be easy to reuse code
2. Big classes are more difficult to change
3. Big classes are harder to read
4. Dont code for situations that you wont ever need
5. Dont create unneeded complexity
6. However, **more class files != more complicated** Offshoot of SRP - Small Methods
7. A method should have one purpose (reason to change)
8. Easier to read and write, which means you are less likely to write bugs
9. Write out the steps of a method using plain English method names

The Open/Closed Principle (OCP)

Software entities (classes, modules, functions, etc.) should be open for ex- tension but closed for modification.

When a single change to a program results in a cascade of changes to dependent modules, the design smells of rigidity. OCP advises us to refactor the system so that further changes of that kind will not cause more modifications.

If OCP is applied well, further changes of that kind are achieved by adding new code, not by changing old code that already works.

Anytime you change code, you have the potential to break it

Modules that conform to OCP have two primary attributes.

1. They are open for extension. This means that the behavior of the module can be extended. As the requirements of the application change, we can extend the module with new behaviors that satisfy those changes. We are able to change what the module does.
2. They are closed for modification. Extending the behavior of a module does not result in changes to the source, or binary, code of the module.

The normal way to extend the behavior of a module is to make changes to the source code of that module. Modules behaviours can be changed without modifying the code by means of abstractions. In object-oriented programming language (OOPL), it is possible to create abstractions that are fixed and yet represent an unbounded group of possible behaviors.

The abstractions are abstract base classes, and the unbounded group of possible behav- iors are represented by all the possible derivative classes. It is possible for a module to manipulate an abstraction.

Such a module can be closed for modification, since it depends on an abstraction that is fixed. Yet the behavior of

that module can be extended by creating new derivatives of the abstraction It is possible for a module to manipulate an abstraction. Such a module can be closed for modification, since it depends on an abstraction that is fixed. Yet the behavior of that module can be extended by creating new derivatives of the abstraction. Two Design Patterns used to enforce OCP are Template Method and Strategy.

These two patterns are the most common ways of satisfying OCP. They represent a clear sep- aration of generic functionality from the detailed implementation of that functionality. Will cover more of these in the next chapter on Design Patterns and Refactoring.

The Open/Closed Principle is at the heart of object-oriented design. Con- formance to this principle is what yields the greatest benefits claimed for object-oriented technology: flexibility, reusability, and maintainability.

The Liskov Substitution Principle (LSP)

Subclasses should be substitutable for their base classes. The Super class and the sub class should be substitutable and make sense when used.

We mentioned that OCP is the most important of the class category principles. We can think of the Liskov Substitution Principle (LSP) as an extension to OCP. In order to take advantage of LSP, we must adhere to OCP because violations of LSP also are violations of OCP, but not vice versa.

In its simplest form, LSP is difficult to differentiate from OCP, but a subtle difference does exist. OCP is centered around abstract coupling. LSP, while also heavily dependent on abstract coupling, is in addition heavily dependent on preconditions and postcondi- tions, which is LSPs relation to Design by Contract, where the concept of preconditions and postconditions was formalized.

A precondition is a contract that must be satisfied before a method can be invoked. A postcondition, on the other hand, must be true upon method completion. If the precondition is not met, the method shouldnt be invoked, and if the postcondition is not met, the method shouldnt return.

The Liskovs Substitution Principle provides a guideline to sub-typing any existing type. Stated formally it reads:

If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behaviour of P is unchanged when o1 is substituted for o2 then S is a subtype of T.

In a simpler term, if a program module is using the reference of a Base class, then it should be able to replace the Base class with a Derived class without affecting the functioning of the program module.

The Dependency-Inversion Principle (DIP)

Depend upon abstractions. Do not depend upon concretions.

The Dependency Inversion Principle (DIP) formalizes the concept of abstract coupling and clearly states that we should couple at the abstract level, not at the concrete level.

Two classes are tightly coupled if they are linked together and are dependent on each other Tightly coupled classes can not work independent of each other It Makes changing one class difficult because it could launch a wave of changes through tightly coupled classes

In our own designs, attempting to couple at the abstract level can seem like overkill at times. Pragmatically, we should apply this principle in any situation where were unsure whether the implementation of a class may change in the future.

But in reality, we encounter situations during development where we know exactly what needs to be done.

Requirements state this very clearly, and the probability of change or extension is quite low. In these situations, adherence to DIP may be more work than the benefit realized.

At this point, there exists a striking similarity between DIP and OCP. In fact, these two principles are closely related. Fundamentally, DIP tells us how we can adhere to OCP. Or, stated differently, if OCP is the desired end, DIP is the means through which we achieve that end. While this statement may seem obvious, we commonly violate DIP in a certain situation and dont even realize it.

Abstract coupling is the notion that a class is not coupled to another concrete class or class that can be instantiated. Instead, the class is coupled to other base, or abstract, classes. This abstract class can be either a class with the abstract modifier or a Java interface data type.

Regardless, this concept actually is the means through which LSP achieves its flexibility, the mechanism required for DIP, and the heart of OCP. The OCP is the guiding principle for good Object- Oriented design.

The design typically has two aspects with it. One, you design an application module to make it work and second, you need to take care whether your design, and thereby your application, module is reusable, flexible and robust. The OCP tells you that the software module should be open for extension but closed for modification.

As you might have already started thinking, this is a very high level statement and the real problem is how to achieve this. Well, it comes through practice, experience and constant inspection of any piece of design, understanding that how it performs and works tackles with the expanding re- quirements of the application. But even though you are a new designer as student, you can follow certain principles to make sure that your design is a good one.

The Dependency Inversion Principle helps you make your design OCP compliant. Formally stated, the principle makes two points: 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.

The Interface Segregation Principle (ISP)

Many specific interfaces are better than a single, general interface. This principle deals with the disadvantages of ”fat” interfaces. Classes whose interfaces are not cohesive have ”fat” interfaces. In other words, the interfaces of the class can be broken up into groups of methods. Each group serves a different set of clients. Thus, some clients use one group of methods, and other clients use the other groups.

Any interface we define should be highly cohesive. In Java, we know that an interface is a reference data type that can have method declarations, but no implementation. In essence, an interface is an abstract class with all abstract methods.

As we define our interfaces, it becomes important that we clearly understand the role the interface plays within the context of our application. In fact, interfaces provide flexibility: They allow objects to assume the data type of the interface.

Consequently, an interface is simply a role that an object plays at some point throughout its lifetime. It follows, rather logically, that when defining the operation on an interface, we should do so in a manner that doesnt accommodate multiple roles.

Therefore, an interface should be responsible for allowing an object to assume a SINGLE ROLE, assuming the class of which that object is an instance implements that interface. **Composite Reuse Principle (CRP)**

Favor polymorphic composition of objects over inheritance. The **Composite Reuse Principle (CRP)** prevents us from making one of the most catas- trophic mistakes that contribute to the demise of an object-oriented system: using in- heritance as the primary reuse mechanism.

Inheritance can be thought of as a generalization over a specialization relationship That is, a class higher in the

inheritance hierarchy is a more general version of those inherited from it. In other words, any ancestor class is a partial descriptor that should define some default characteristics that are applicable to any class inherited from it.

Any time we have to override default behavior defined in an ancestor class, we are saying that the ancestor class is not a more general version of all of its descendants but actually contains descriptor characteristics that make it too specialized to serve as the ancestor of the class in question.

Therefore, if we choose to define default behavior on an ancestor, it should be general enough to apply to all of its descendents. In practice, its not uncommon to define a default behavior in an ancestor class. However, we should still accommodate CRP in our relationships.

Principle of Least Knowledge (PLK)

For an operation O on a class C, only operations on the following objects should be called:itself, its parameters, objects it creates, or its contained instance objects.

The Principle of Least Knowledge (PLK) is also known as the Law of Demeter. The basic idea is to avoid calling any methods on an object where the reference to that object is obtained by calling a method on another object.

Instead, this principle recommends we call methods on the containing object, not to obtain a reference to some other object, but instead to allow the containing object to forward the request to the object we would have formerly obtained a reference to.

The primary benefit is that the calling method doesnt need to understand the structural makeup of the object its invoking methods upon. The obvious disadvantage associated with PLK is that we must create many methods that only forward method calls to the containing classes internal components. This can contribute to a large and cumbersome public interface.

An alternative to PLK, or a variation on its implementation, is to obtain a reference to an object via a method call, with the restriction that any time this is done, the type of the reference obtained is always an interface data type.

This is more flexible because we arent binding ourselves directly to the concrete imple- mentation of a complex object, but instead are dependent only on the abstractions of which the complex object is composed.

This is how many classes in Java typically resolve this situation.

Software Packaging Principles

In this section, we explore the principles of design that help us split a large software system into packages.

As software applications grow in size and complexity, they require some kind of highlevel organization. Classes are convenient unit for organizing small applications but are too finely grained to be used as the sole organizational unit for large applications. Something ”larger” than a class is needed to help organize large applications. That something is called a package, or a component.

This section outlines six principles for managing the contents and relationships between components. The first three, principles of package cohesion, help us allocate classes to packages. The last three principles govern package coupling and help us determine how packages should be interrelated.

The last two principles also describe a set of dependency management metrics that allow developers to measure and characterize the dependency structure of their designs.

Principles of Component Cohesion: Granularity

The principles of component cohesion help developers decide how to partition classes into components. These principles depend on the fact that at least some of the classes and their interrelationships have been discovered. Thus, these principles take a bottom-up view of partitioning.

The Reuse/Release Equivalence Principle (REP)

The granule of reuse is the granule of release.

REP states that the granule of reuse, a component, can be no smaller than the granule of release. Anything that we reuse must also be released and tracked. It is not realistic for a developer to simply write a class and then claim that it is reusable.

Reusability comes only after a tracking system is in place and offers the guarantees of notification, safety, and support that the potential reusers will need.

Whenever a client class wishes to use the services of another class, we must reference the class offering the desired services. If the class offering the service is in the same package as the client, we can reference that class using the simple name. If, however, the service class is in a different package, then any references to that class must be done using the class fully qualified name, which includes the name of the package. Any Java class may reside in only a single package.

Therefore, if a client wishes to utilize the services of a class, not only must we reference the class, but we must also explicitly make reference to the containing package. Failure to do so results in compile- time errors. Therefore, to deploy any class, we must be sure the containing package is deployed. Because the package is deployed, we can utilize the services offered by any public class within the package.

While we may presently need the services of only a single class in the containing package, the services of all classes are available to us.

Consequently, our unit of release is our unit of reuse, resulting in the Release Reuse Equivalency Principle (REP). This leads us to the basis for this principle, and it should now be apparent that the packages into which classes are placed have a tremendous impact on reuse. Careful consideration must be given to the allocation of classes to packages.

The Common Reuse Principle (CReP)

Classes that arent reused together should not be grouped together.

If we need the services offered by a class, we must import the package containing the necessary classes.

As we stated previously in our discussion of REP (Release Reuse Equivalency Principle), when we import a package, we also may utilize the services offered by any public class within the package. In addition, changing the behavior of any class within the service package has the potential to break the client.

Even if the client doesnt directly reference the modified class in the service package, other classes in the service package being used by clients may reference the modified class. This creates indirect dependencies between the client and the modified class that can be the cause of mysterious behavior. We can state the following: If a class is dependent on another class in a different package, then it is dependent on all classes in that package, albeit indirectly. This principle has a negative connotation.

It doesnt hold true that classes that are reused together should reside together, depend- ing on CCP. Even though classes may always be reused together, they may not always change together. In striving to adhere to CCP, separating a set of classes based on their likelihood to change together should be given careful consideration.

Of course, this impacts REP because now multiple packages must be deployed to use this functionality. Experience tells us that adhering to one of these principles may impact the ability to adhere to another. Whereas REP and Common Reuse Principle (CReP) emphasize reuse, CCP emphasizes maintenance.

The Common Closure Principle (CCP)

Classes that change together, belong together.

The basis for the Common Closure Principle (CCP) is rather simple. Adhering to fundamental programming best practices should take place throughout the entire system. Functional cohesion emphasizes well-written methods that are more easily maintained. Class cohesion stresses the importance of creating classes that are functionally sound and dont cross responsibility boundaries.

And package cohesion focuses on the classes within each package, emphasizing the overall services offered by entire packages.

During development, when a change to one class may dictate changes to another class, its preferred that these two classes be placed in the same package.

Conceptually, CCP may be easy to understand; however, applying it can be difficult because the only way that we can group classes together in this manner is when we can predictably determine the changes that might occur and the effect that those changes might have on any dependent classes.

Predictions often are incorrect or arent ever realized.

Regardless, placement of classes into respective packages should be a conscious decision that is driven not only by the relationships between classes, but also by the cohesive nature of a set of classes working together.

Principles of Component Coupling: Stability

The next three principles deal with the relationships between components. Here again, we will run into the tension between developability and logical design. The forces that impinge on the architecture of a component structure are technical, political, and volatile.

The Acyclic Dependencies Principle (ADP)

The dependencies between packages must form no cycles.

Cycles among dependencies of the packages composing an application should almost always be avoided.

Packages should form a Directed Acyclic Graph (DAG). Acyclic Dependencies Principle (ADP) is important from a deployment perspective.

Along with packages being reusable and maintanable, the should also be deployable as well. Just as in the class design, the package design should have defined dependencies so that it is deployment- friendly.

The Stable-Dependencies Principle (SDP)

Depend in the direction of stability.

Stability implies that an item is fixed, permanent, and unvarying. Attempting to change an item that is stable is more difficult than inflicting change on an item in a less stable state. Aside from poorly written code, the degree of coupling to other packages has a dramatic impact on the ease of change.

Those packages with many incoming dependencies have many other components in our application dependent on them.

These more stable packages are difficult to change because of the far-reaching conse- quences the change may have throughout all other dependent packages. On the other hand, packages with few incoming dependencies are easier to change.

Those packages with few incoming dependencies most likely will have more outgoing dependencies. A package with no incoming or outgoing dependencies is useless and isnt part of an application because it has no relationships.

Therefore, packages with fewer incoming, and more outgoing dependencies, are less stable. Stability metrics Stability is calculated by counting the number of dependencies that enter and leave that component.

These counts allow us to calculate the positional stability of the component:

Ca (afferent couplings): The number of classes outside this component that depend on classes within this component

Ce (efferent couplings): The number of classes inside this component that depend on classes outside this component

Formula: I(instability) = Ce/(Ce +Ca)

This metric has the range [0,1]. I = 0 indicates a maximally stable component. I = 1 indicates a maximally unstable component.

The Ca and Ce metrics are calculated by counting the number of classes outside the component in question that have dependencies on the classes inside the component in question.

The Stable-Abstractions Principle (SAP)

Stable packages should be abstract packages.

One of the greatest benefits of object orientation is the ability to easily maintain our systems. The high degree of resiliency and maintainability is achieved through abstract coupling. By coupling concrete classes to abstract

classes, we can extend these abstract classes and provide new system functions without having to modify existing system structure.

Consequently, the means through which we can depend in the direction of stability, and help ensure that these more depended-upon packages exhibit a higher degree of stability, is to place abstract classes, or interfaces, in the more stable packages.

We can state the following: More stable packages, containing a higher number of abstract classes, or interfaces, should be heavily depended upon. Less stable packages, containing a higher number of concrete classes, should not be heavily depended upon. Any packages containing all abstract classes with no incoming dependencies are utterly useless.

On the other hand, packages containing all concrete classes with many incoming depen- dencies are extremely difficult to maintain.

Measuring abstraction

The A metric is a measure of the abstractness of a component. Its value is simply the ratio of abstract classes in a component to the total number of classes in the component, where **Nc** is the number of classes in the component. **Na** is the number of abstract classes in the component. Remember, an abstract class is a class with at least one abstract method and cannot be instantiated:

Chapter Exercises

1. Write small program using TDD method to demonstrate the three core principals of Object Oriented Programming
2. In question 1, you wrote code demonstrating inheritance. Modify your code to use an alternative solution to inheritance.
3. There are 7 core software principles and for each write code that violet the principle and then write code that obeys the principle
4. Write code that violets ADP and also write code that corrects the violation.

Chapter 5: Design Patterns and Refactoring

1. Be able to know what design patterns are.
2. From a given list, select the most appropriate pattern for a given scenario and demonstrate its applicability
3. From a list, select the most appropriate pattern for a given scenario. Patterns are limited to those documented in the book - Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides (1995). Design Patterns: Elements of Reusable Object-Oriented Software and are named using the names given in that book.
4. Understand the idea of refactoring and re-factor software code to Design Patterns

Design Patterns

What are Patterns

Design Patterns are not classes, Design Patterns are not frameworks. Instead, Design Patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.

So, Design patterns are formalized solutions to design problems. They describe techniques for maximizing flexibility, extensibility, abstraction, etc. These solutions can typically be translated to code in a straight forward manner.

It is important to note the fact that Design Patterns makes our life easy by giving guidelines to apply in a given context and for a specific problem.

Elements of Patterns

Pattern Name

This is more than just a handle for referring to the pattern. Each name adds to a designers vocabulary. It enables the discussion of design at a higher abstraction.

The Problem

This gives a detailed description of the problem addressed by the pattern. It describes when to apply a pattern, often with a list of preconditions.

The Solution

This describes the elements that make up the design, their relation- ships,responsibilities, and collaborations. It does not describe a concrete solution. In- stead a template to be applied in many situations

The consequences

This describes the results and tradeoffs of applying the pattern. It is critical for evalu- ating design alternatives Typically include impact on flexibility, extensibility, or porta- bility, Space and Time tradeoffs, Language and Implementation issues

Design Patterns Templates

1. Pattern Name and Classification
    - Creational
    - Structural
    - Behavioural
2. Intent Also Known As Motivation and Applicability
3. Structure and Participants
4. Collaborations
5. Consequences
6. Implementation
7. Sample Code
8. Known Uses
9. Related Patterns

Sample Documentation of a Design Pattern

- Pattern and Classification

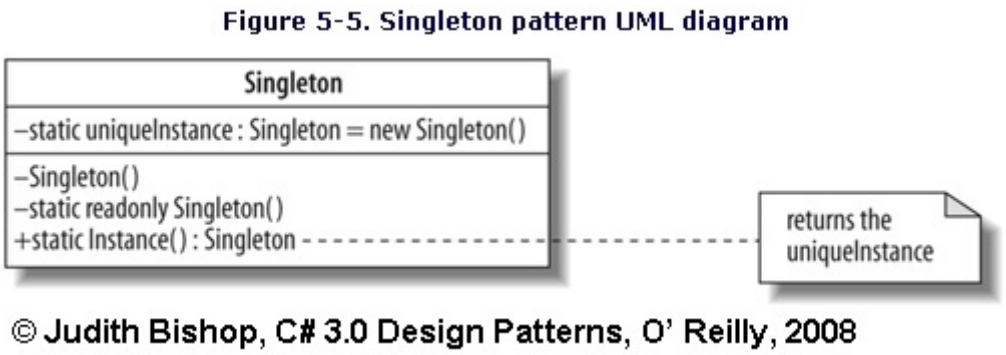
```
Singleton and Crational Pattern
```

- Intent Also Known As Motivation and Applicability

Ensure a class has only one instance, and provide a global point of access to it Some classes represent objects where multiple instances do not make sense or can lead to a security risk (e.g. Java security managers) and memory or heap overload

Use the Singleton pattern when there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point, when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code

- Structure and Participants



Just the Singleton class

- Collaborations

Clients access a Singleton instance solely through Singletons Instance operation

- Consequences

Controlled access to sole instance. Reduced name space (versus global variables) Permits a variable number of instances (if desired)

- Impleentation

```
public class SingletonExample {

    private static SingletonExample singletonExample = null;

    private SingletonExample() {
    }

    public static SingletonExample getInstance() {
        if (singletonExample == null) {
            singletonExample = new SingletonExample();
        }
        return singletonExample;
    }

    public void sayHello() {
        System.out.println("Hello");
    }
```

```
}
```

- Sample Code Usage

```
public class MainDriver {

    public static void main(String[] args) {
        SingletonExample singletonExample = SingletonExample.getInstance();

        singletonExample.sayHello();
    }

}
```

- Known Uses

The problem addressed by the singleton pattern arises when you have an object that you don’t want duplicated throughout an application, either because it represents a real-world resource (such as a printer or server) or because you want to consolidate a set of related activities in one place. When it comes to real-world resources, the ability to create new objects that represent printers or servers is nonsensical because creating an object doesn’t magically put new hardware into place.

- Related Patterns

Creational Patterns

GOF Patterns

There are totally 23 design patterns in GoF (Gang of Four). All these 23 design patterns are mapped to 3 main categories:

1. Creational patterns : Makes Object creation/instantiation job easy.
2. Structural patterns : Makes Objects available to other class by changing the in- terfaces or contract.
3. Behavioral patterns: Deals with object state changes and it’s interaction with other classes/objects.

Creational Patterns

The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling thisobject creation.

Creational patterns involve the construction of new objects by often hiding the constructors in the classes being created, and provides alternate methods to return instances of the desired class.

All the creational patterns define the best possible way in which an object can be createdconsidering reuse and changeability. These describes the best way to handle instantiation.

Hard coding the actual instantiation is a pitfall and should be avoided if reuse and changeability are desired. In such scenarios, we can make use of creational patterns to give this a more general and flexible approach.

1. Singleton
2. Factory Method
3. Abstract Factory
4. Builder Pattern
5. Prototype

Singleton

The singleton pattern ensures that only one object of a given type exists in the application.

The singleton pattern can be used to manage objects that represent realworld resources or to encapsulate a shared resource.

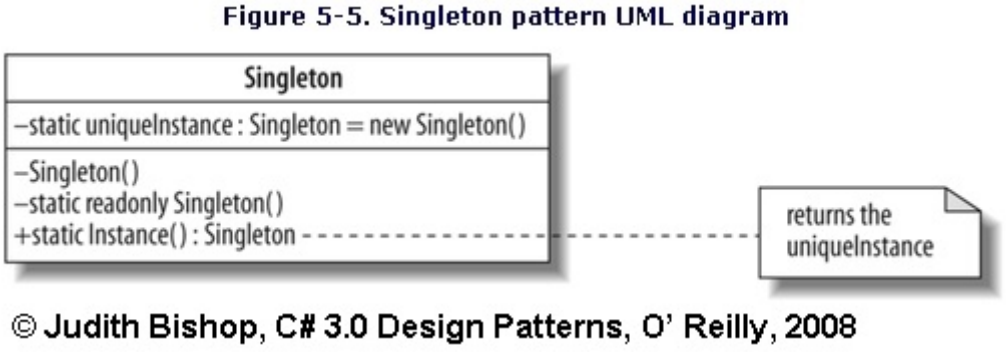
The singleton pattern should be used when creating further objects doesn't increase the number of realworld resources available or when you want to consolidate an activity such as logging.

The singleton pattern isn't useful if there are not multiple components that require access to a shared resource or if there are no objects that represent realworld resources in the application.

The pattern has been correctly implemented when there is only one instance of a given type and when that instance cannot be copied and cloned and when further instances cannot be created.

The main pitfalls are using reference types (which can be copied). The singleton pattern usually requires some protections against concurrent use, which is a common source of problems.

Structure



Example Code

The Sample code shows the creation of a Calculator Singleton

```
package zm.hashcode.sample.creational.singleton;

/**
 * Created by hashcode on 2015/02/26.
 */
public class Calculator {
    private static Calculator calculator = null;

    private Calculator() {
    }

    public static Calculator getInstance() {
        if (calculator == null) {
            calculator = new Calculator();
        }
        return calculator;
    }

    public int add(int a, int b) {
        return a + b;
    }

    public int multiply(int a, int b) {
        return a * b;
    }

    public int substract(int a, int b) {
        return a - b;
    }
}
```

Factory Method

The factory method pattern is used when there is a choice to be made between classes that implement a common base class. This pattern allows implementation subclasses to provide specializations without requiring the components that rely on them to know any details of those classes and how they relate to each other.

The factory method pattern selects an implementation class to satisfy a calling component's request without requiring the component to know anything about the implementation classes or the way they relate to one another.

This pattern consolidates the logic that decides which implementation class is selected and prevents it from being diffused throughout the application. This also means that calling components rely only on the top-level protocol or base class and do not need any knowledge about the implementation classes or the process by which they are selected.

Use this pattern when you have several classes that implement a common interface or that are derived from the same base class.

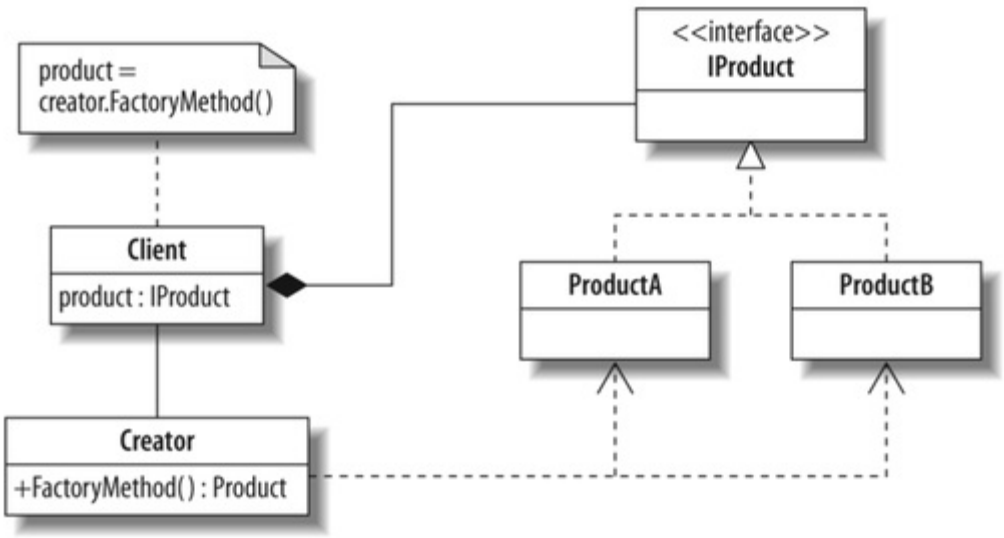
Do not use this pattern when there is no common interface or shared base class because this pattern works by having the calling component rely on only a single type.

This pattern is implemented correctly when the appropriate class is instantiated without the calling client knowing which class was used or how it was selected.

The factory method pattern is often combined with the singleton.

Structure

Figure 5-3. Factory Method pattern UML diagram



Example Code

The Sample code addresses the problem of creating different types Employees that have a common parent lass called Emplyee

The Emplaoyee Common Class

```
package zm.hashcode.sample.creational.factory;

/**
 * Created by hashcode on 2015/02/26.
 */
public interface Employee {
    public abstract String role();
}
```

The Employee Factory Method

```
package zm.hashcode.sample.creational.factory;

/**
 * Created by hashcode on 2015/02/26.
 */
public class EmployeeFactory {
    private static EmployeeFactory employeeFactory = null;

    private EmployeeFactory() {
    }

    public static EmployeeFactory getEmployeeFactoryInstance() {
        if (employeeFactory == null)
            return new EmployeeFactory();
        return employeeFactory;
    }

    public Employee getEmployeeRole(String employee) {
        if ("Lecturer".equalsIgnoreCase(employee)) {
            return new Lecturer();
        } else {
            return new Secretary();
        }
    }
}
```

The Lecturer Class

```
package zm.hashcode.sample.creational.factory;

/**
 * Created by hashcode on 2015/02/26.
 */
public class Lecturer implements Employee{
    @Override
    public String role() {
        return "Lecturer at CPUT";
    }
}
```

The Secretary class

```
package zm.hashcode.sample.creational.factory;

/**
 * Created by hashcode on 2015/02/26.
 */
public class Secretary implements Employee{
    @Override
    public String role() {
        return "Secretary at CPUT";
    }
}
```

Abstract Factory

The abstract factory pattern allows a calling client to create a group of related objects. The pattern hides the details of which classes are used to create the objects and the reason why they were selected from the calling client. This pattern is similar to the factory method but presents the calling client with a set of objects.

The calling client doesn't know which classes are used to create the objects or why they were selected, which makes it possible to change the classes that are used without needing to change the clients that consume them.

Use this pattern when you need to ensure that multiple compatible objects are used by a calling component without the component needing to know which objects are able to work together.

Do not use this pattern to create a single object; the factory method pattern is a simpler alternative that should be used instead.

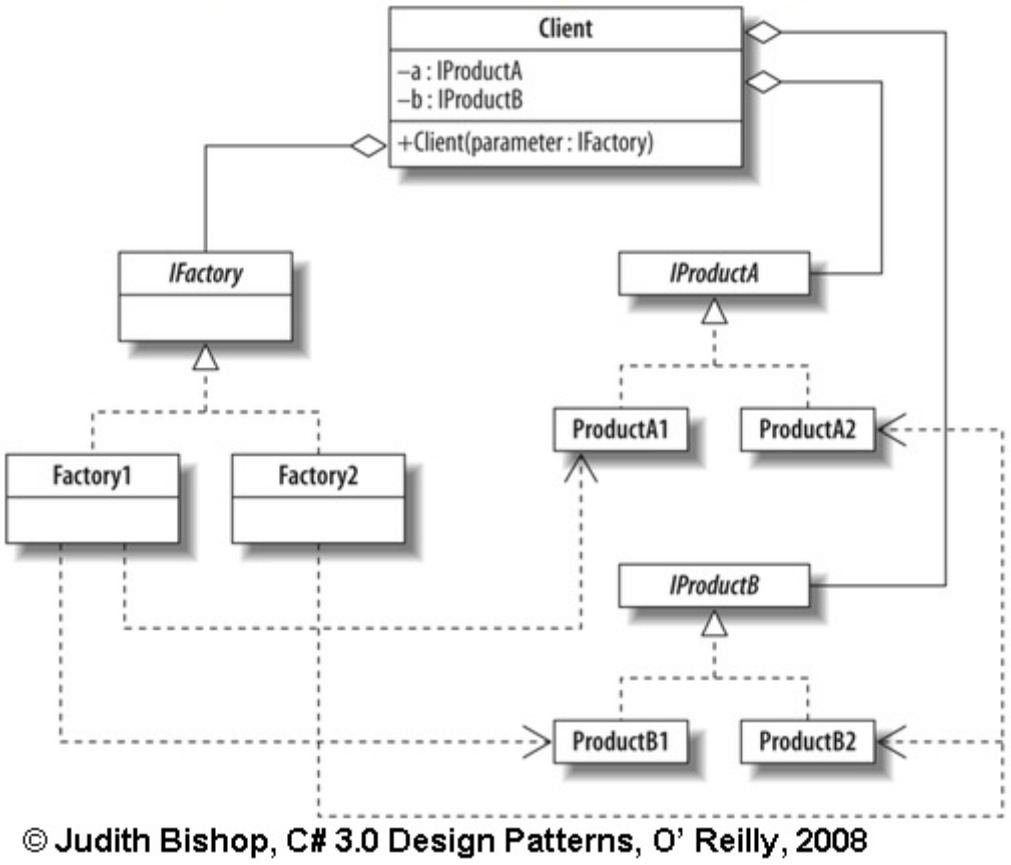
This pattern is implemented correctly when a calling component receives a set of objects without knowing which classes were used to instantiate them. The calling component should be able to access the object's functionality only through the protocols they implement or the base classes from which they are derived.

The main pitfall is to leak details of the classes that are used to the calling component, either creating a dependency on the decision making process that selects classes or creating a dependency on specific classes.

The factory method pattern is a simpler pattern when only a single object is required. The abstract factory method is often combined with the singleton.

Structure

Figure 6-2. Abstract Factory pattern UML diagram



Example Code

This Sample code shows the creation of groups of Year Subjects with their names

The Subject Interface

```
package zm.hashcode.sample.creational.abstractfactory;

/**
 * Created by hashcode on 2015/02/26.
 */
public interface Subject {
    public abstract String getSubjectName();
}
```

The Subject factory

```
package zm.hashcode.sample.creational.abstractfactory;

/**
 * Created by hashcode on 2015/02/26.
 */
public interface SubjectsFactory {
    public abstract Subject getSubjectName(String subjectCode);
}
```

Second Year Subjects

```
package zm.hashcode.sample.creational.abstractfactory;

/**
```

```
 * Created by hashcode on 2015/02/26.
 */
public class SecondYearDS implements Subject{
    @Override
    public String getSubjectName() {
        return " Development Software 2";
    }
}


package zm.hashcode.sample.creational.abstractfactory;

/**
 * Created by hashcode on 2015/02/26.
 */
public class SecondYearTP implements Subject{
    @Override
    public String getSubjectName() {
        return "Technical Programming 1";
    }
}
```

Third Year Subject

```
package zm.hashcode.sample.creational.abstractfactory;

/**
 * Created by hashcode on 2015/02/26.
 */
public class ThirdYearDS implements Subject {

    @Override
    public String getSubjectName() {
        return "Development Software 2";
    }
}


package zm.hashcode.sample.creational.abstractfactory;

/**
 * Created by hashcode on 2015/02/26.
 */
public class ThirdYearTP implements Subject{
    @Override
    public String getSubjectName() {
        return "Technical Programming 2";
    }
}
```

The Second year Factory method

```
package zm.hashcode.sample.creational.abstractfactory;

/**
 * Created by hashcode on 2015/02/26.
 */
public class SecondYearSubjectsFactory implements SubjectsFactory {
    @Override
    public Subject getSubjectName(String subjectCode) {
        // Complete Implementation based on the Factory method Example
        return null;
    }
}
```

Third Year Factory method

```
package zm.hashcode.sample.creational.abstractfactory;
```

```
/**
 * Created by hashcode on 2015/02/26.
 */
public class ThirdYearSubjectsFactory  implements SubjectsFactory{

    @Override
    public Subject getSubjectName(String subjectCode) {
        // Complete Implementation based on the Factory method Example
        return null;
    }
}
```

Builder Pattern

The builder pattern puts the logic and default configuration values required to create an object into a builder class. This allows calling ctients to create objects with minimal configuration data and without needing to know the default values that will be used to create the object.

This pattern makes it easier to change the default configuration values used to create an object and to change the class from which instances are created.

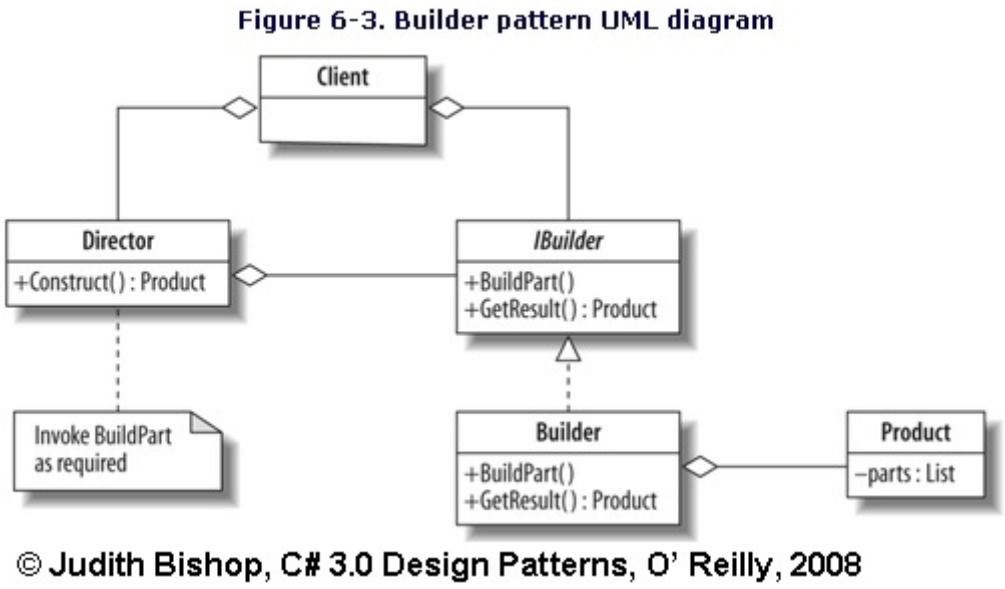
Use this pattern when a complex configuration process is required to create an object and you don't want the default configuration values to be disseminated throughout the application.

Don't use this pattern when every data value required to create an object will be different for each instance.

The calling client can create objects by providing just the data values for which there are no default values

This pattern can be combined with the factory method or abstract factory patterns to change the implementation class used to create the object based on the configuration data provided by the calling client.

Structure



Example Code

The sample code code shows how one would build a Diploma course combo using a bulder Pattern

The Course Template

```
package zm.hashcode.sample.creational.builder;

/**
 * Created by hashcode on 2015/02/26.
 */
public class DiplomaCourse {
    private String technicalPrograming;
    private String informationSystems;
    private String internetPrograming;
    private String developmentSoftware;

    public String getTechnicalPrograming() {

        return technicalPrograming;
    }

    public void setTechnicalPrograming(String technicalPrograming) {
        this.technicalPrograming = technicalPrograming;
    }

    public String getInformationSystems() {

        return informationSystems;
    }

    public void setInformationSystems(String informationSystems) {
        this.informationSystems = informationSystems;
    }

    public String getInternetPrograming() {
        return internetPrograming;
    }

    public void setInternetPrograming(String internetPrograming) {
        this.internetPrograming = internetPrograming;
    }

    public String getDevelopmentSoftware() {
        return developmentSoftware;
    }

    public void setDevelopmentSoftware(String developmentSoftware) {
        this.developmentSoftware = developmentSoftware;
    }

    @Override
    public String toString() {
        return "DiplomaCourse{" +
                "buildTechnicalPrograming='" + technicalPrograming + '\'' +
                ", buildInformationSystems='" + informationSystems + '\'' +
                ", buildInternetPrograming='" + internetPrograming + '\'' +
                ", buildDevelopmentSoftware='" + developmentSoftware + '\'' +
                '}';
    }
}
```

The Course Builder

```
public interface DiplomaCourseBuilder {
    public void buildTechnicalPrograming();

    public void buildInformationSystems();

    public void buildInternetPrograming();

    public void buildDevelopmentSoftware();

    public DiplomaCourse getDiplomaCourse();
}
```

The Course Director

```
package zm.hashcode.sample.creational.builder;

/**
 * Created by hashcode on 2015/02/26.
 */
public class DiplomaCourseDirector {
    private DiplomaCourseBuilder diplomaCourseBuilder= null;

    public DiplomaCourseDirector(DiplomaCourseBuilder diplomaCourseBuilder) {
        this.diplomaCourseBuilder = diplomaCourseBuilder;
    }

    public void constructDiplomaCourse(){
            diplomaCourseBuilder.buildDevelopmentSoftware();
            diplomaCourseBuilder.buildInformationSystems();
            diplomaCourseBuilder.buildTechnicalPrograming();
            diplomaCourseBuilder.buildInternetPrograming();
    }

    public DiplomaCourse getDiplomaCourse() {
        return diplomaCourseBuilder.getDiplomaCourse();
    }
}
```

And Now we can build the Couserses starting with Second Year Combo

```
package zm.hashcode.sample.creational.builder;

/**
 * Created by hashcode on 2015/02/26.
 */
public class SecondYearDiplomaCourseBuilder implements DiplomaCourseBuilder {
    private DiplomaCourse diplomaCourse;

    public SecondYearDiplomaCourseBuilder() {
        diplomaCourse = new DiplomaCourse();
    }

    @Override
    public void buildTechnicalPrograming() {
        diplomaCourse.setTechnicalPrograming("Technical Programming 1");


    }

    @Override
    public void buildInformationSystems() {
        diplomaCourse.setInformationSystems("Information System 2");

    }

    @Override
    public void buildInternetPrograming() {
        diplomaCourse.setInternetPrograming("Internet Programing 2");

    }

    @Override
    public void buildDevelopmentSoftware() {
        diplomaCourse.setDevelopmentSoftware("Development Software 2");
    }

    @Override
    public DiplomaCourse getDiplomaCourse() {
        return diplomaCourse;
    }
}
```

The Third Year Course

```
package zm.hashcode.sample.creational.builder;
```

```
/**
 * Created by hashcode on 2015/02/26.
 */
public class ThirdYearDiplomaCourseBuilder implements DiplomaCourseBuilder{
    private DiplomaCourse diplomaCourse;

    public ThirdYearDiplomaCourseBuilder() {

        diplomaCourse = new DiplomaCourse();
    }

    @Override
    public void buildTechnicalPrograming() {
        diplomaCourse.setTechnicalPrograming("Technical Programing 2");

    }

    @Override
    public void buildInformationSystems() {
        diplomaCourse.setInformationSystems("Information System 3");

    }

    @Override
    public void buildInternetPrograming() {
        diplomaCourse.setInternetPrograming("Internet Programming 3");

    }

    @Override
    public void buildDevelopmentSoftware() {
        diplomaCourse.setDevelopmentSoftware("Development Software 3");

    }

    @Override
    public DiplomaCourse getDiplomaCourse() {
        return diplomaCourse;
    }
}
```

Prototype

The prototype pattern creates new objects by copying an existing object, known as the prototype.

The main benefit is to hide the code that creates objects from the clients that use them; this means that clients don't need to know which class is required to create a new object, don't need to know the details of initializers, and don't need to change when subclasses are created and instantiated. This pattern can also be used to avoid repeating expensive initialization each time a new object of a specific type is created.

This pattern is useful when you are writing a client that needs to create new instances of objects without creating a dependency on the class initializer.

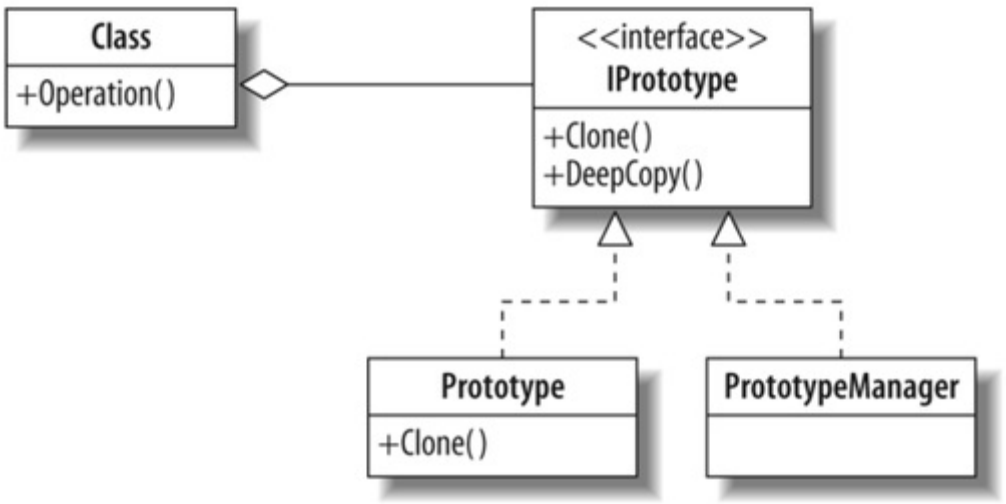
To test for an effective implementation of this pattern, change the initializer for the class used for the prototype object and check to see whether a corresponding change is required in the client that creates clones. As a second test, create a subclass of the prototype's class and ensure that the client can clone it without requiring any changes.

The main pitfall is selecting the wrong style of copying when cloning the prototype object. There are two kinds of copying available—shallow and deep—and it is important to select the correct kind for your application.

The most closely related pattern is the object template pattern

Structure

Figure 5-1. Prototype pattern UML diagram



Example Code

The Clone Interface

```
package zm.hashcode.sample.creational.prototype;

/**
 * Created by hashcode on 2015/02/26.
 */
public interface CloneObject {
    public CloneObject makeCopy();
}
```

The Clonable Lecturer

```
package zm.hashcode.sample.creational.prototype;

/**
 * Created by hashcode on 2015/02/26.
 */
public class Lecturer implements CloneObject{
    private String name;

    public Lecturer(String name) {
        this.name = name;
    }

    @Override
    public CloneObject makeCopy() {
        return new Lecturer(name);
    }

    @Override
    public String toString() {
        return "Lecturer{" +
                "name='" + name + '\'' +
                '}';
    }
}
```

The Cloneable Subject

```
package zm.hashcode.sample.creational.prototype;

/**
```

```
 * Created by hashcode on 2015/02/26.
 */
public class Subject implements CloneObject{
    private String subjectCode;
    public Subject(String subjectCode) {
        this.subjectCode = subjectCode;
    }
    @Override
    public CloneObject makeCopy() {
        return new Subject(subjectCode);
    }
    @Override
    public String toString() {
        return "Subject{" +
                "subjectCode='" + subjectCode + '\'' +
                '}';
    }
}
```

Structural Patterns

In software engineering, structural design patterns are design patterns that ease the design by identifying a simple way to realize relationships between entities. A structural design pattern serves as a blueprint for how different classes and objects are combined to form larger structures. Unlike creational patterns, which are mostly different ways to fulfill the same fundamental purpose, each structural pattern has a different purpose.

Structural Patterns describe how objects and classes can be combined to form structures. We distinguish between object patterns and class patterns. The difference is that class patterns describe relationships and structures with the help of inheritance. Object patterns, on other hand, describe how objects can be associated and aggregated to form larger, more complex structures. In some sense, structural patterns are similar to the simpler concept of data structures.

However, structural design patterns also specify the methods that connect objects, not merely the references between them. Furthermore, data structures are fairly static entities. They only describe how data is arranged in the structure. A structural design pattern also describes how data moves through the pattern. Struc- tural class patterns use inheritance to combine the interfaces or implementations of multiple classes.

Structural object patterns use object composition to combine the implementations of multiple objects. They can combine the interfaces of all the composed objects into one unified interface or they can provide a completely new interface.

1. Adapter pattern
2. Composite pattern
3. Proxy pattern
4. Flyweight Pattern
5. Facade Pattern
6. Bridge Pattern
7. Decorator Pattern

Adapter pattern

The adapter pattern allows two client with incompatible APIs to work together by introducing an adapter that maps from one component to the other.

This pattern allows you to integrate clients for which you cannot modify the source code into your application. This is a common problem when you use a third-party framework or when you are consuming the output from another project.

Use this pattern when you need to integrate a component that provides similar functionality to other components in the application but that uses an incompatible API to do so.

Do not use this pattern when you are able to modify the source code of the component that you want to integrate or when it is possible to migrate the data provided by the component directly into your application.

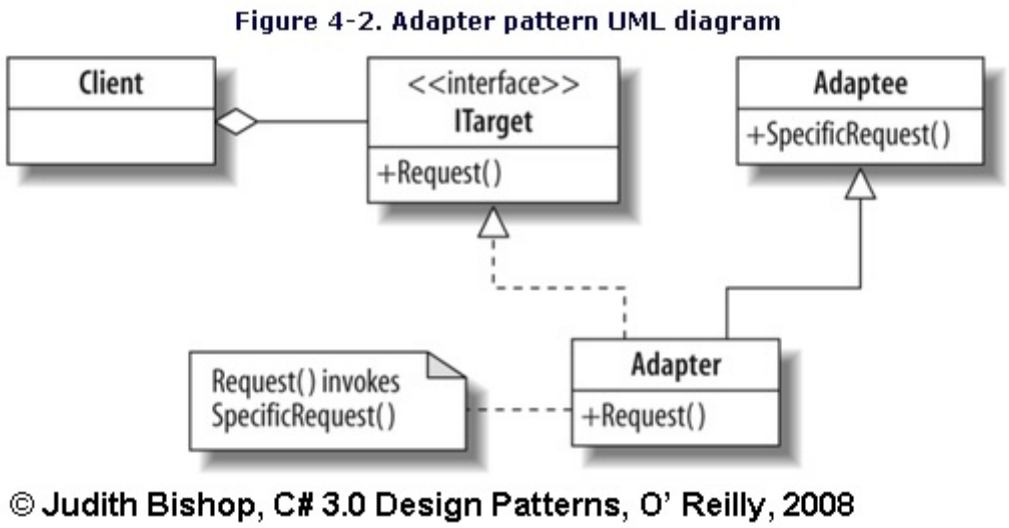
The pattern is implemented correctly when the adapter allows the component to be integrated into the application without requiring modification of the application or the component.

The only pitfall is trying to extend the pattern to force integration of a component that does not provide the functionality intended by the API for which it is being adapted.

Many of the structural patterns have similar implementations but different intents. Ensure that you select the correct

pattern

Structure



Sample Code

The sample code is for a typical adaptor plug that provides different voltages. First we need a Socket, Adapter and Voltage

The Socket

```
package zm.hashcode.sample.structural.adapter;

/**
 * Created by hashcode on 2015/02/27.
 */
public class Socket {
    public Voltage getVoltage(){
        return new Voltage(240);
    }
}
```

The Socket Adapter

```
package zm.hashcode.sample.structural.adapter;

/**
 * Created by hashcode on 2015/02/27.
 */
public interface SocketAdapter {
    public Voltage get240Volt();

    public Voltage get12Volt();

    public Voltage get3Volt();

}
```

The Voltage

```
package zm.hashcode.sample.structural.adapter;

/**
 * Created by hashcode on 2015/02/27.
```

```
 */
public class Voltage {
    private int volts;

    public Voltage(int volts) {
        this.volts = volts;
    }

    public int getVolts() {
        return volts;
    }

    public void setVolts(int volts) {
        this.volts = volts;
    }
}
```

The Class Implementation

```
package zm.hashcode.sample.structural.adapter.classadaptor;

import zm.hashcode.sample.structural.adapter.Socket;
import zm.hashcode.sample.structural.adapter.SocketAdapter;
import zm.hashcode.sample.structural.adapter.Voltage;

/**
 * Created by hashcode on 2015/02/27.
 */
public class SocketClassAdapter extends Socket implements SocketAdapter {

    @Override
    public Voltage get240Volt() {
        return getVoltage();
    }

    @Override
    public Voltage get12Volt() {
        Voltage v= getVoltage();
        return convertVolt(v,10);
    }

    @Override
    public Voltage get3Volt() {
        Voltage v= getVoltage();
        return convertVolt(v,40);
    }

    private Voltage convertVolt(Voltage v, int i) {
        return new Voltage(v.getVolts()/i);
    }

}
```

The Object Implementation

```
package zm.hashcode.sample.structural.adapter.objectadaptor;

import zm.hashcode.sample.structural.adapter.Socket;
import zm.hashcode.sample.structural.adapter.SocketAdapter;
import zm.hashcode.sample.structural.adapter.Voltage;

/**
 * Created by hashcode on 2015/02/27.
 */
public class SocketObjectAdapter implements SocketAdapter {

    //Using Composition for adapter pattern
    private Socket sock = new Socket();

    @Override
    public Voltage get240Volt() {
```

```
        return sock.getVoltage();
    }

    @Override
    public Voltage get12Volt() {
        Voltage v = sock.getVoltage();
        return convertVolt(v, 10);
    }

    @Override
    public Voltage get3Volt() {
        Voltage v = sock.getVoltage();
        return convertVolt(v, 40);
    }

    private Voltage convertVolt(Voltage v, int i) {
        return new Voltage(v.getVolts() / i);
    }
}
```

Composite pattern

The composite pattern is not as broadly applicable as some of the other design patterns, but it remains an important pattern because of the way it applies consistency to a data structure that contains different types of object.

The composite pattern allows a tree of individual objects and collections of objects to be treated consistently.

The consistency that the composite pattern brings means that components that operate on the tree structure are simpler and do not need to have knowledge of the different objects types that are in use.

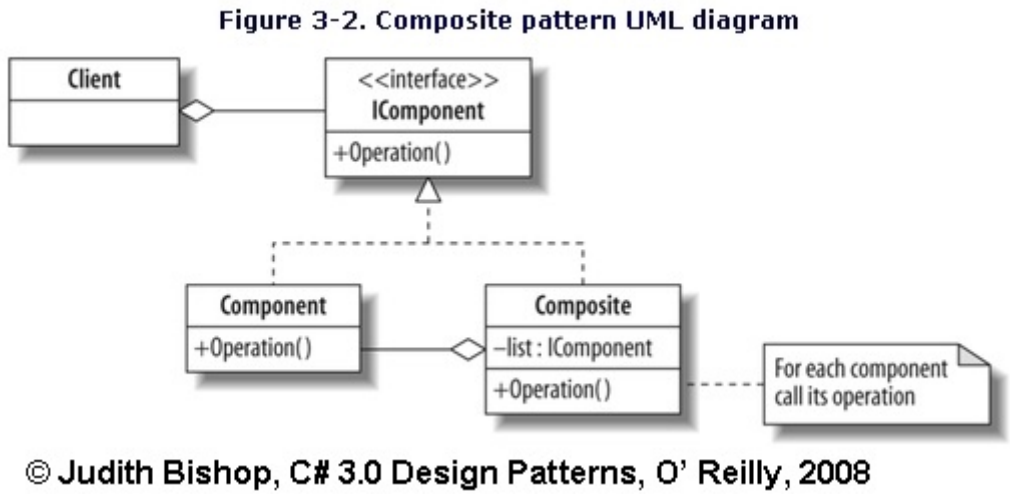
Use this pattern when you have a tree structure that contains leaf nodes and collections of objects.

This pattern is applicable only to tree data structures.

The pattern is implemented correctly when components that use the tree structure can treat all of the objects it contains using the same type.

This pattern is best suited to tree structures that are not modified once they have been created. Adding support for modifying the tree undermines the benefit of the pattern.

Structure



Example Code

Proxy Pattern

The proxy pattern defines an object—the proxy—that represents some other resource, such as another object or a remote service. Calling components operate on the proxy, which in turn operates on the underlying resource.

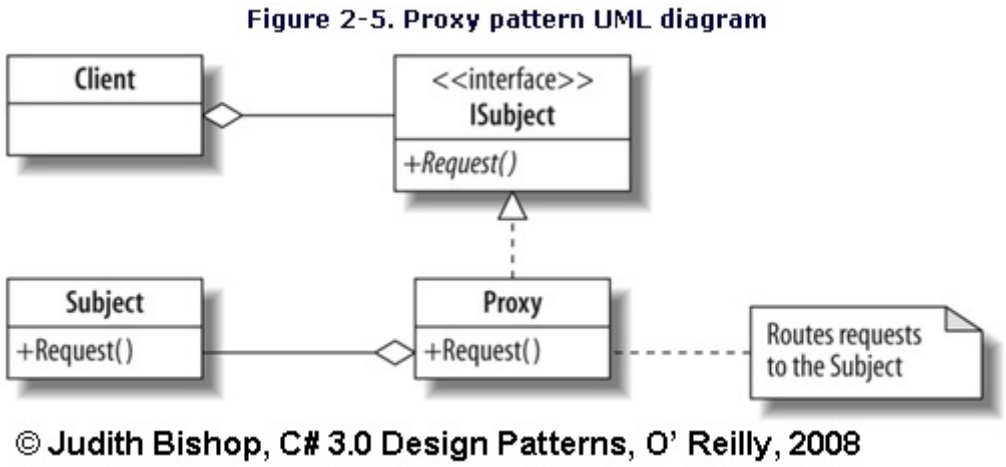
Proxies allow close control over the way that the underlying resource is accessed, which is useful when you need to intercept and adapt operations.

Proxies are used in three main situations: to define an interface to a remote resource such as a web page or RESTful service, to manage the execution of expensive operations, and to restrict access to the methods and properties of other objects.

Do not use this pattern when the problem falls outside of the three situations. Instead, use one of the other structural patterns.

The pattern is implemented correctly when the proxy object can be used to perform operations on the resource it represents.

Structure



Example Code

Flyweight Pattern

The flyweight pattern is applied when a number of similar objects all rely on the same set of data values. Rather than create a new set of data valued for each of the objects, the flyweight pattern shares one set between all of them, minimizing the amount of memory required to store the data and the amount of work required to create them.

So the flyweight pattern shares common data objects between multiple calling clients.

The greatest advantage of the Flyweight Pattern is that it reduces the amount of memory needed to create the data objects required by the calling components and the amount of work required to create them. The impact of implementing the pattern increases with the number of calling clients that share the data.

Use this pattern when you are able to identify and isolate sets of identical data objects that are used by calling clients.

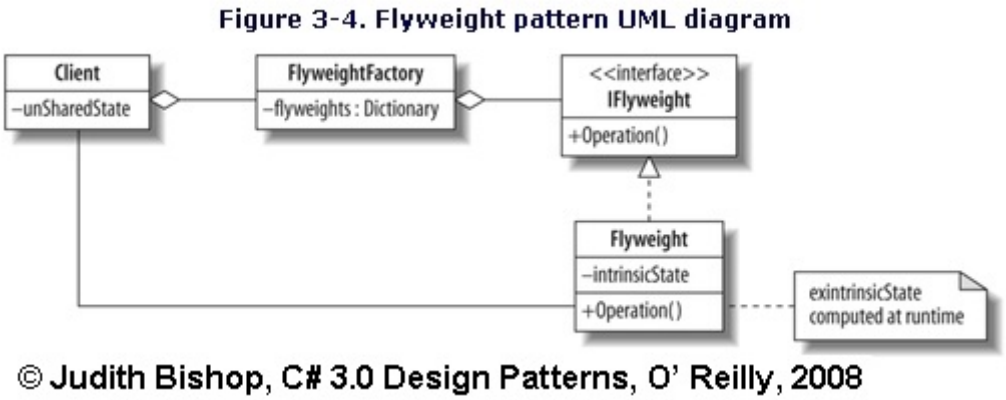
Do not use this pattern if there is no shared data or if the number of shared data objects is small and easy to create.

The pattern said to have been implemented correctly when all of the calling clients rely on the same set of

immutable shared data objects and have their own individual state data. The calling clients should be able to concurrently modify their copied data safely and not be able to modify the shared data at all.

The common pitfalls include inadvertently creating more than one set of shared data objects, not protecting against concurrent operations on the copied data, allowing the shared data to be modified, and over-optimizing the creation of shared objects.

Structure



Example Code

Facade Pattern

The façade pattern is used to simplify the API presented by one or more classes so that common tasks can be performed more easily and the complexity required to use the API is consolidated in one part of the application. So in short, the façade pattern simplifies the use of complex APIs to perform common tasks.

The complexity required to use an API is consolidated into a single class, which minimizes the impact of changes in the API and simplifies the clients that consume the API functionality.

Use the façade pattern when you are working with classes that need to be used together but that don't have compatible APIs.

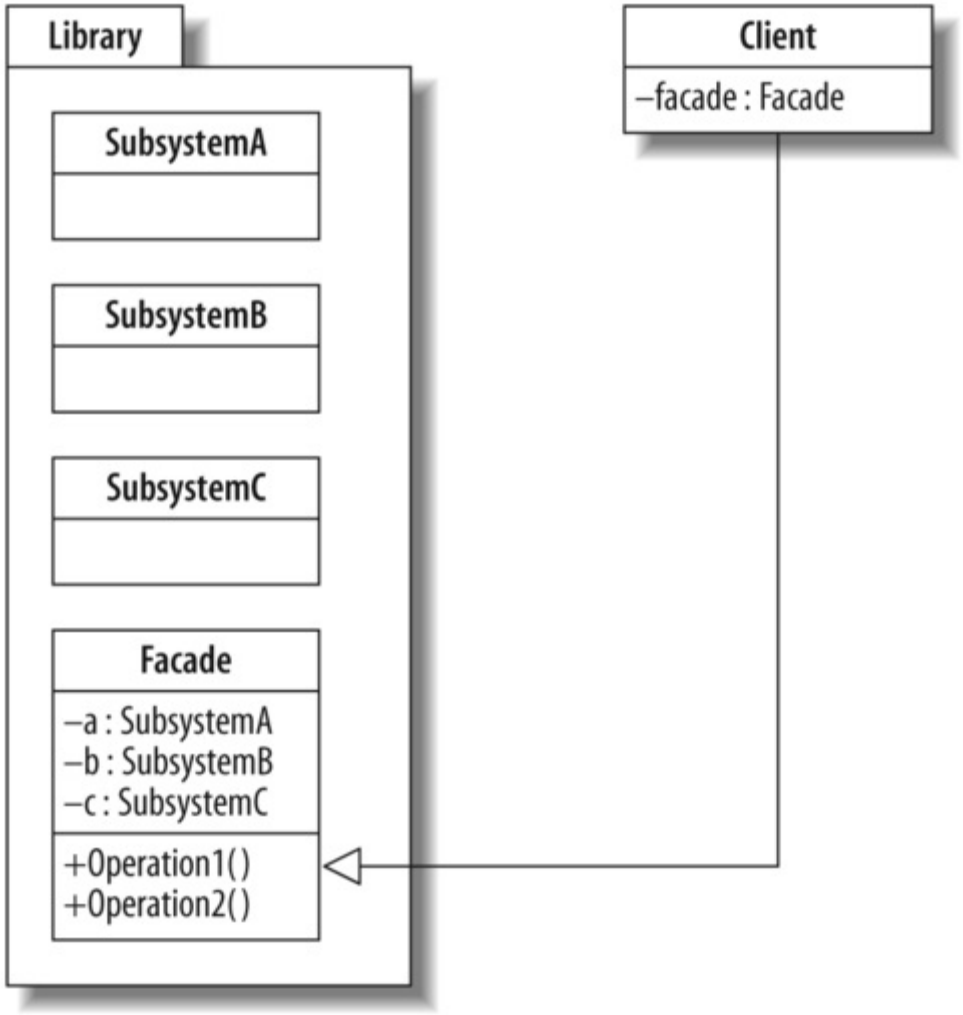
Do not use the façade pattern when integrating single clients into the application; use the adapter pattern instead.

The façade pattern is implemented when common tasks can be performed without calling clients having any dependency on the underlying objects or their supporting data types.

The pitfall when implementing the façade pattern is to leak details of the underlying objects. This means that the calling clients are still dependent on the underlying classes or supporting types and will require modification when they change.

Structure

Figure 4-5. Façade pattern UML diagram



Example Code

Bridge Pattern

The bridge pattern separates an abstraction from its implementation so that either can be changed without a corresponding change in the other. More commonly, the bridge pattern is used to resolve a problem known as an exploding class hierarchy, which usually arises through repeated but poorly thought-out refactoring and requires an ever-increasing number of classes to add new features to the application.

When the bridge pattern is applied to the exploding class hierarchy problem, the benefit is that adding a new feature to the application requires only a single class. More broadly, the pattern isolates the impact of a change when an abstraction or its implementation changes.

Use this pattern to resolve the exploding class hierarchy problem or to bridge between one API and another.

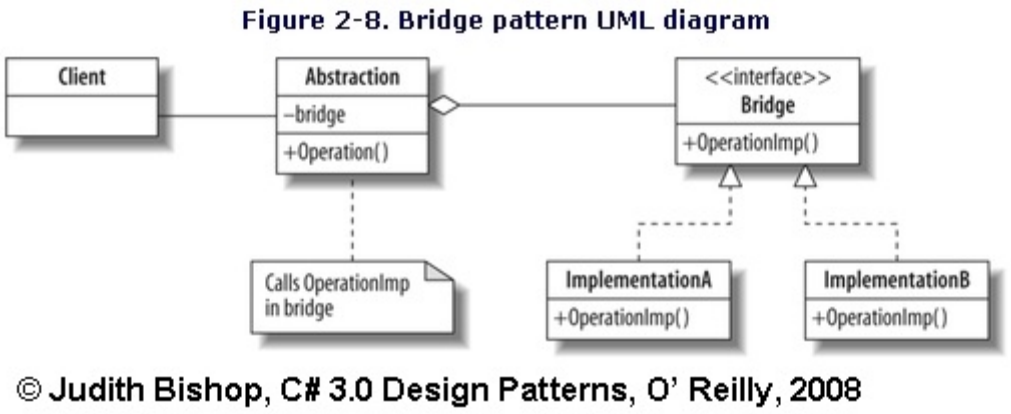
Do not use this pattern when attempting to integrate third-party components; use the adapter pattern

In the case of the exploding class hierarchy problem, the pattern is correctly implemented when adding a new feature or when support for a new platform can be done with a single class. More broadly, the pattern is implemented correctly when you can change an abstraction such as an interface or a closure signature without having to make a corresponding change in its implementation.

The exploding class hierarchy will not be resolved if the common code is not separated from the platform-specific

code.

Structure



Example Code

Decorator Pattern

The decorator pattern allows the behavior of individual objects to be changed without requiring changes to the classes that are used to create them or the clients that consume them.

The changes in behavior defined with the decorator pattern can be combined to create complex effects without needing to create large numbers of subclasses.

Use this pattern when you need to change the behavior of objects without changing the class they are created from or the client that use them.

Do not use this pattern when you are able to change the class that creates the objects you want to modify. It is usually simpler and easier to modify the class directly.

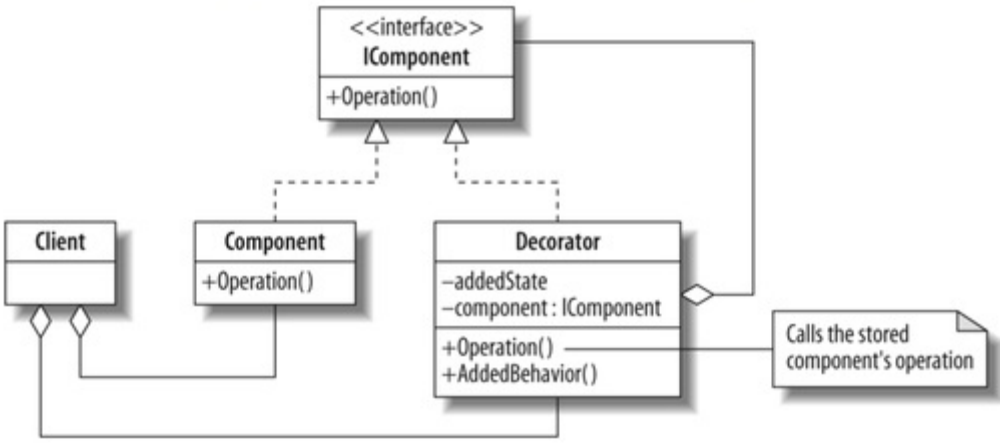
The pattern is said to have been implemented correctly when you can select some of the objects created from a class to be modified without affecting all of them and without requiring changes to the class.

The main pitfall is implementing the pattern so that it affects all of the objects created from a given class rather than allowing changes to be applied selectively.

A less common pitfall is implementing the pattern so that it has hidden side effects that are not related to the original purpose of the objects being modified.

Structure

Figure 2-2. Decorator pattern UML diagram



Example Code

Behavioural Patterns

In software engineering, behavioral design patterns are design patterns that identify common communication patterns between objects and realize these patterns. By doing so, these patterns increase flexibility in carrying out this communication.

A behavioral pattern explains how objects interact. It describes how different objects and classes send messages to each other to make things happen and how the steps of a task are divided among different objects. It describes how different objects work together to accomplish a task Where creational patterns mostly describe a moment of time (the instant of creation), and structural patterns describe a more or less static structure, behavioral patterns describe a process or a flow.

The interactions between cooperating objects should be such that they are communi- cating while maintaining as loose coupling as possible. The loose coupling is the key to n-tier architectures. In this, the implementation and the client should be loosely coupled in order to avoid hard-coding and dependencies.

Behavioral class patterns use inheritance, subclassing, and polymorphism to adjust the steps taken during a process. Behavioral class patterns focus on changing the exact algorithm used or task performed depending on circumstances.

1. Template Method Pattern
2. Mediator Pattern
3. Chain of Responsibility Pattern
4. Observer Pattern
5. Strategy Pattern
6. Command Pattern
7. State Pattern
8. Visitor Pattern
9. Iterator Pattern
10. Memento Pattern
11. Interpreter Pattern

Template Method Pattern

The template method pattern allows for individual steps in an algorithm to be changed, which is useful when you are writing classes with default behavior that you want to allow to be changed by other developers.

The template method pattern allows specific steps in an algorithm to be replaced by implementations provided by a third-party, either by specifying functions as closures or by creating a subclass.

This pattern is useful when you are writing frameworks that you want to allow other developers to extend and customize.

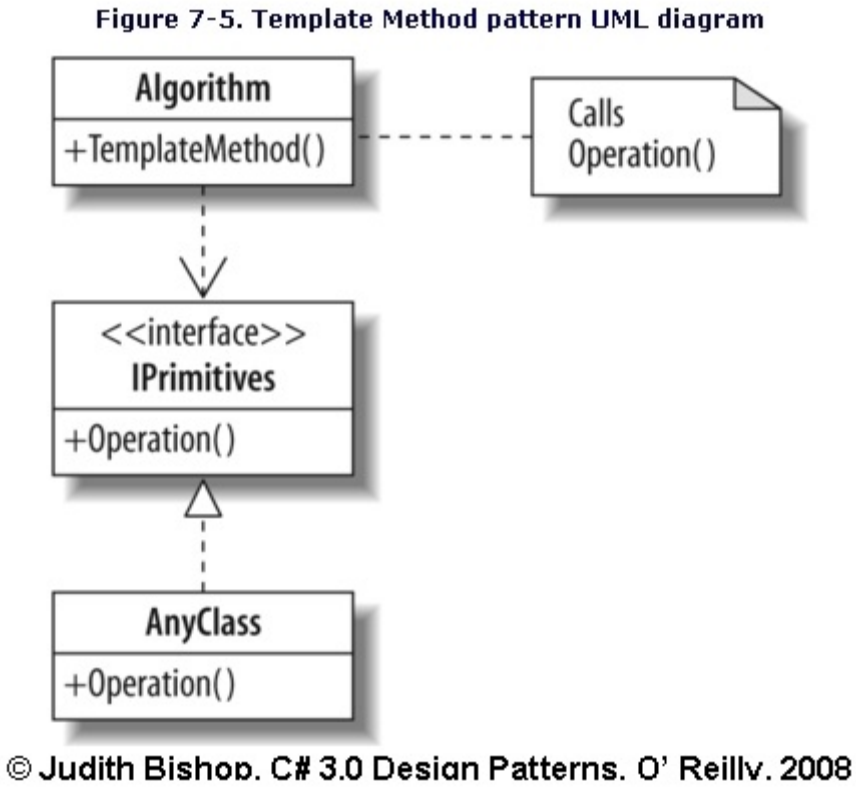
Use this pattern to selectively permit steps in any algorithm to be changed without modifying the original class.

Do not use this pattern if the entire algorithm can be changed. See the other patterns in this part of the book for alternatives.

This pattern is implemented correctly when selected steps in an algorithm can be changed without modifying the class that defines the algorithm.

This pattern has similar goals to the strategy and visitor patterns

Structure



Sample Code

```
```

Mediator Pattern

The mediator pattern is used to simplify and rationalize the communication between groups of objects.

The mediator pattern simplifies peer-to-peer communication between objects by introducing a mediator object that acts as a communications broker between the objects.

Instead of having to keep track of and communicate with of all of its peers individually, an object just deals with the mediator.

Use this pattern when you are dealing with a group of objects that need to communicate freely between one another.

Don't use this pattern if you have one object that needs to send notifications to a range of disparate objects; use the observer pattern.

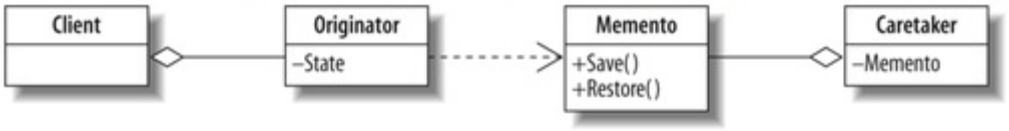
The mediator pattern is implemented correctly when each object deals only with the mediator and has no direct knowledge of its peers.

It is important that the mediator not provide peers with access to one another so that they might become interdependent.

This pattern is closely related to—and often combined with—the observer pattern

Structure

Figure 10-8. Memento pattern UML diagram



Sample Code

```

```

Chain of Responsibility Pattern

The chain of responsibility pattern organizes sequentially a set of objects that may be able to take responsibility for a request from a calling component. The sequence of objects is referred to as a chain, and each object in the chain is asked to take responsibility for the request. The request moves along the chain until one of the objects takes responsibility or the end of the chain is reached

The chain of responsibility allows objects that can process requests to be ordered into a preferential sequence that can be reordered, extended, or reduced without any impact on the calling component, which has no insight into the objects that comprise the chain.

Use this pattern when there are several objects that can handle a request, only one of which should be used.

Do not use this pattern when there is only one object that can handle a request or when the calling client needs to select the object.

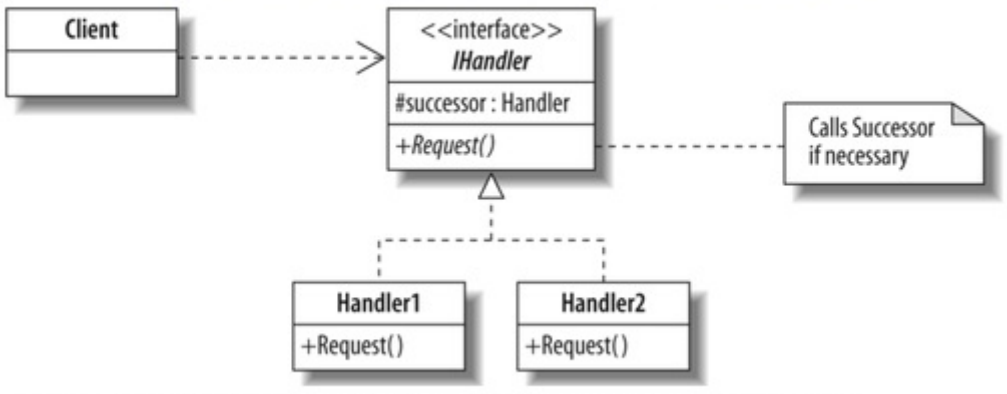
The pattern is implemented correctly when the set of objects that can take responsibility for a request are arranged sequentially and each is offered the chance to take responsibility in turn. The individual objects in the chain have no knowledge of one another other than the next link in the chain.

The pitfall is leaking details of the objects in the chain, either to one another or to the calling component.

The chain of responsibility pattern shares some common concepts with the command pattern

Structure

Figure 8-2. Chain of Responsibility pattern UML diagram



Sample Code

This Sample code show the application of a Chain of responsibility to a grade system that can be solved using a series of if..lelse statement and so violating the Open Closed Principle. The Chain of Resposibility pattern is used enforce OCP

Below is the

```
package zm.hashcode.sample.behavioural.chainofresponsibility;

/**
 * Created by hashcode on 2015/03/02.
 */
public abstract class GradeHandler {
    GradeHandler successor;

    public void setSuccessor(GradeHandler successor) {
        this.successor = successor;
    }

    public abstract String handleRequest(int request);

}
```

And then the different Handlers for Grades A, B, Cand D

Grade A

```
package zm.hashcode.sample.behavioural.chainofresponsibility;

/**
 * Created by hashcode on 2015/03/02.
 */
public class GradeAHandler extends GradeHandler{
    @Override
    public String handleRequest(int request) {
        if(request > 75){
            return "A";
        }else{
                return successor.handleRequest(request);
        }
    }
}
```

Grade B

```
package zm.hashcode.sample.behavioural.chainofresponsibility;

/**
 * Created by hashcode on 2015/03/02.
 */
public class GradeBHandler extends GradeHandler {
    @Override
    public String handleRequest(int request) {
        if(request > 70 & request < 75){
            return "B";
        }else{
                return successor.handleRequest(request);
        }
    }
}
```

Grade C

```
package zm.hashcode.sample.behavioural.chainofresponsibility;

/**
 * Created by hashcode on 2015/03/02.
 */
public class GradeCHandler extends GradeHandler {
    @Override
    public String handleRequest(int request) {
        if(request > 65 & request < 70){
            return "C";
        }else{
            return successor.handleRequest(request);
        }
    }
}
```

Grade D

```
package zm.hashcode.sample.behavioural.chainofresponsibility;

/**
 * Created by hashcode on 2015/03/02.
 */
public class GradeDHandler extends GradeHandler {
    @Override
    public String handleRequest(int request) {
        if(request > 60 & request < 65){
            return "D";
        }else{
            return successor.handleRequest(request);
        }
    }
}
```

The chain is created at the client and if a new grade is needed. The Old Grades are not modified. Instead a new handler is added to both the ient chain and the code.

Observer Pattern

The observer pattern is used to manage the process by which one object expresses interest in—and receives notification of—changes in another.

The observer pattern allows one object to register to receive notifications about changes in another object without needing to depend on the implementation of that object.

This pattern simplifies application design by allowing objects that provide notifications to do so in a uniform way without needing to know how those notifications are processed and acted on by the recipients.

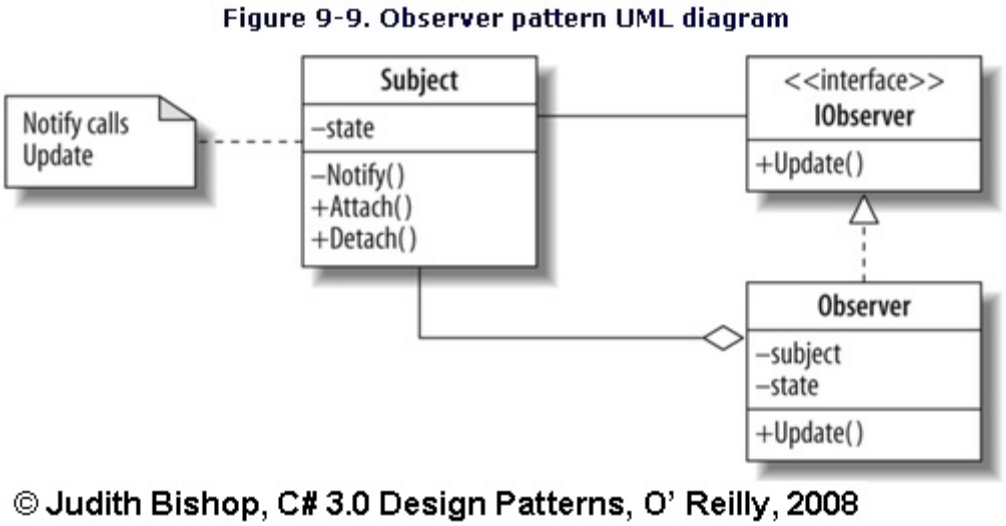
Use this pattern whenever one object needs to receive notifications about changes in another object but where the sender of the notifications does not depend on the recipient to complete its work.

Do not use this pattern unless the sender of the notifications is functionally dependent from the recipients, such that the recipients could be removed from the application without preventing the sender from performing its work.

The observer pattern is implemented correctly when an object can receive notifications without being tightly coupled to the object that sends them.

The biggest pitfall with this pattern is allowing the objects that send and receive notifications to become interdependent

Structure



Sample Code

```
```

Strategy Pattern

The strategy pattern is used to create classes that can be extended without modification, through the application of algorithm objects that conform to a well-defined Interface.

The strategy pattern allows third-party developers to change the behavior of classes without modifying them and can allow low-cost changes to be made in projects that have expensive and lengthy validation procedures for specific classes.

Use this pattern when you need classes that can be extended without being modified.

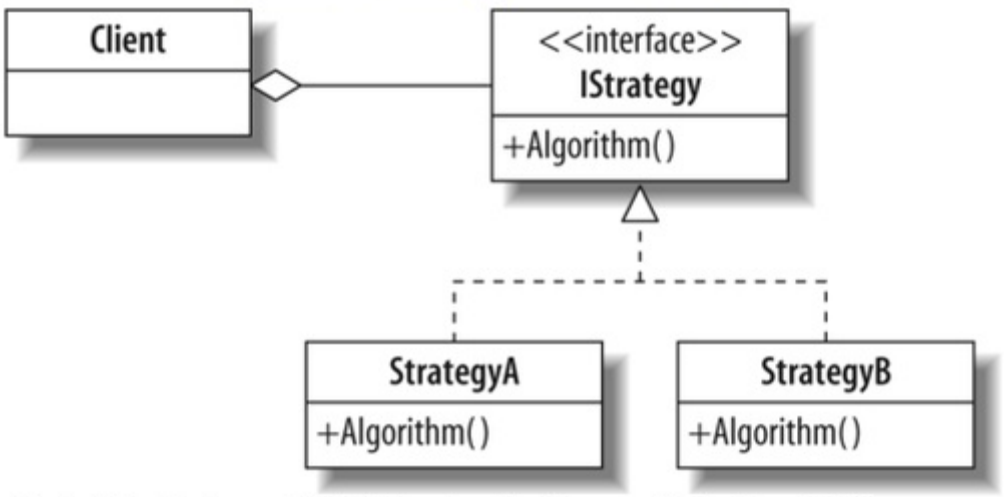
There is no reason to avoid this pattern

The strategy pattern is implemented correctly when you can extend the behavior of a class by defining and applying a new strategy without needing to make any changes to the class itself.

The strategy and visitor patterns are often used together.

Structure

Figure 7-2. Strategy pattern UML diagram



© Judith Bishop, C# 3.0 Design Patterns, O' Reilly, 2008

Sample Code

```
```

Command Pattern

The command pattern provides a mechanism by which details of how to invoke a method can be encapsulated so that the method can be invoked later or by a different client.

The command pattern is used to encapsulate details of how to invokea method on an object in a way that allows the method to be invoked at a different time or by a different client.

There are lots of situations in which using a command is useful, but the most common ones are supporting undo operations and creating macros.

Use this pattern when you want to allow methods to be invoked by client that otherwise have no information about the object that will be used, the method that will be invoked, or the arguments that will be provided.

Do not use this pattern for regular method invocation.

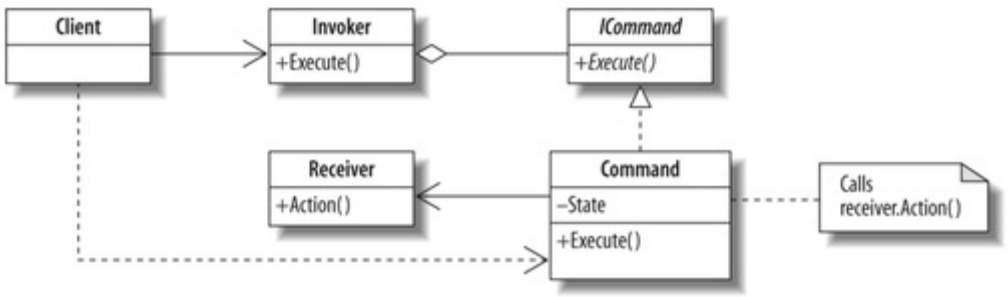
The pattern is implemented correctly when a component can use a command to invoke a method on an object without needing details of that object or the method itself.

The main pitfall is to require the component that executes the command to have knowledge of the method or object that will be used.

The memento pattern provides a model by which snapshots of an object's entire state can be used instead of individual operations.

Structure

Figure 8-6. Command pattern UML diagram



© Judith Bishop, C# 3.0 Design Patterns, O' Reilly, 2008

Sample Code

State Pattern

The State pattern allows an object to alter its behavior when its internal state changes. By using inheritance and letting subclasses represent different states and functionality we can switch during runtime. This is a clean way for an object to partially change its type at runtime.

If we have to change the behavior of an object based on it's state, we can have a state variable in the Object and use if-else condition block to perform different actions based on the state. State pattern is used to provide a systematic and lose-coupled way to achieve this through Context and State implementations.

Context is the class that has a State reference to one of the concrete implementations of the State and forwards the request to the state object for processing. Let's understand this with a simple example.

Suppose we want to implement a TV Remote with a simple button to perform action, if the State is ON, it will turn on the TV and if state is OFF, it will turn off the TV.

Use when we need to define a "context" class to present a single interface to the outside world. By defining a State abstract base class.

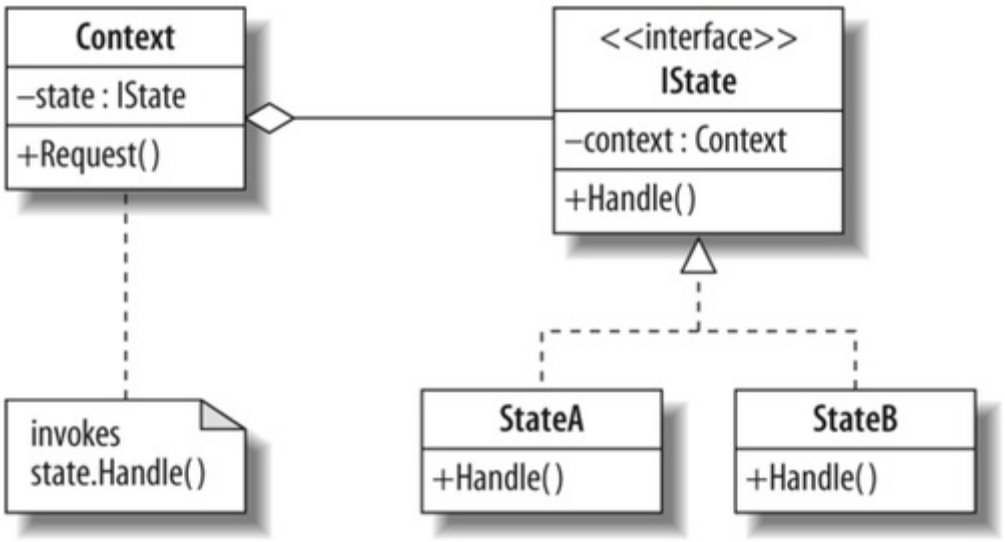
Also use when you want to represent different "states" of a state machine as derived classes of the State base class.

Benefits include Cleaner code when each state is a class instead and Use ofa class to represent a state, not a constant.

Pitfalls include Generating a number of small class objects, but in the process, simplifies and clarifies the program, and eliminating the necessity for a set of long, look-alike conditional statements scattered throughout the code.

Structure

Figure 7-4. State pattern UML diagram



Sample Code

Visitor Pattern

The visitor pattern is similar to the strategy pattern in that it allows the behavior of a class to be extended without modifying its source code or creating a new subclass, except that the visitor pattern is applied to collections of heterogeneous objects.

The visitor pattern allows new algorithms to operate on collections of heterogeneous objects without needing to modify or subclass the collection class.

The visitor pattern is useful when you want to provide collection classes as part of frameworks without requiring third-party developers to modify the source code. This pattern is also useful in projects where modifying core classes triggers expensive testing procedures.

Use this pattern when you have classes that manage collections of mismatched objects and you want to perform operations on them.

There is no need to use this pattern when all of the objects are of the same type or when the collection class can be readily modified.

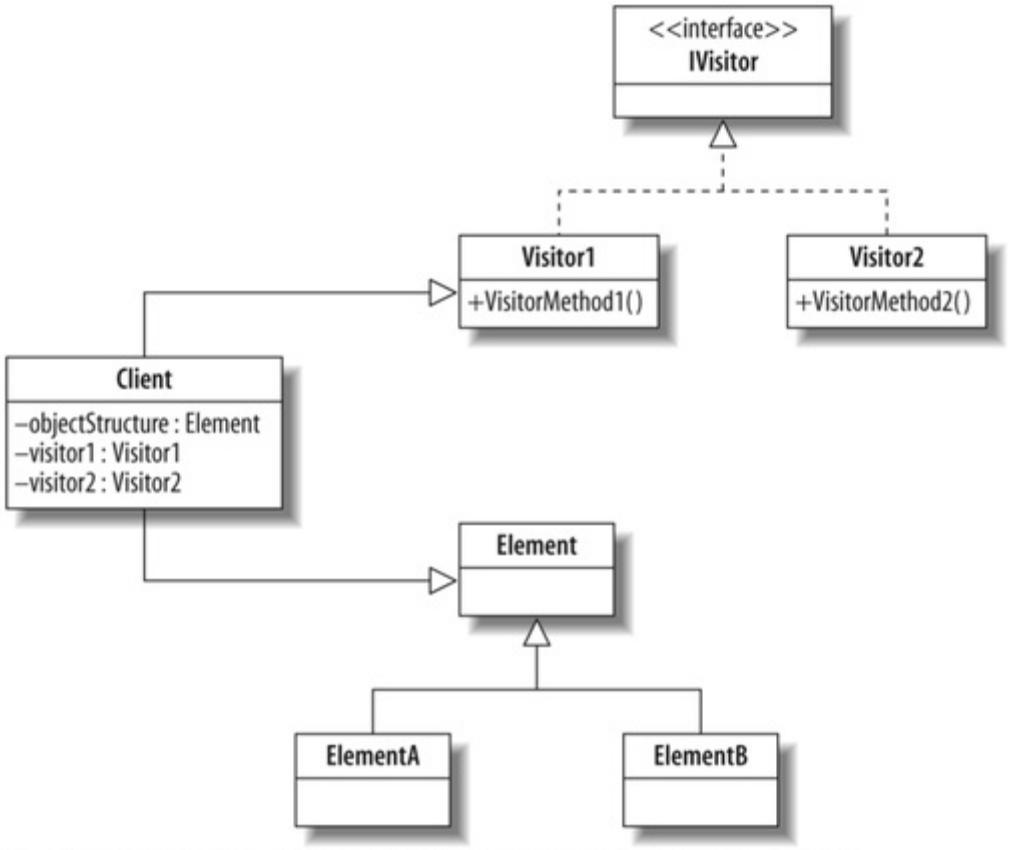
The pattern is implemented correctly when a visitor class can extend the behavior of the collection class by defining methods that handle each type of object in the collection.

The only pitfall is trying to avoid using the double dispatch technique, which I describe in the “Understanding Double Dispatch” sidebar.

The visitor pattern is another way to conform to the open/closed principle supported by the strategy pattern

Structure

Figure 10-3. Visitor pattern UML diagram



Sample Code

```
```

Iterator Pattern

The Iterator design pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Use to access the elements of an aggregate object sequentially. Java's collections like ArrayList and HashMap have implemented the iterator pattern.

The same iterator can be used for different aggregates.

Allows you to traverse the aggregate in different ways depending on your needs.

It encapsulates the internal structure of how the iteration occurs.

Don't need to bloat the your class with operations for different traversals.

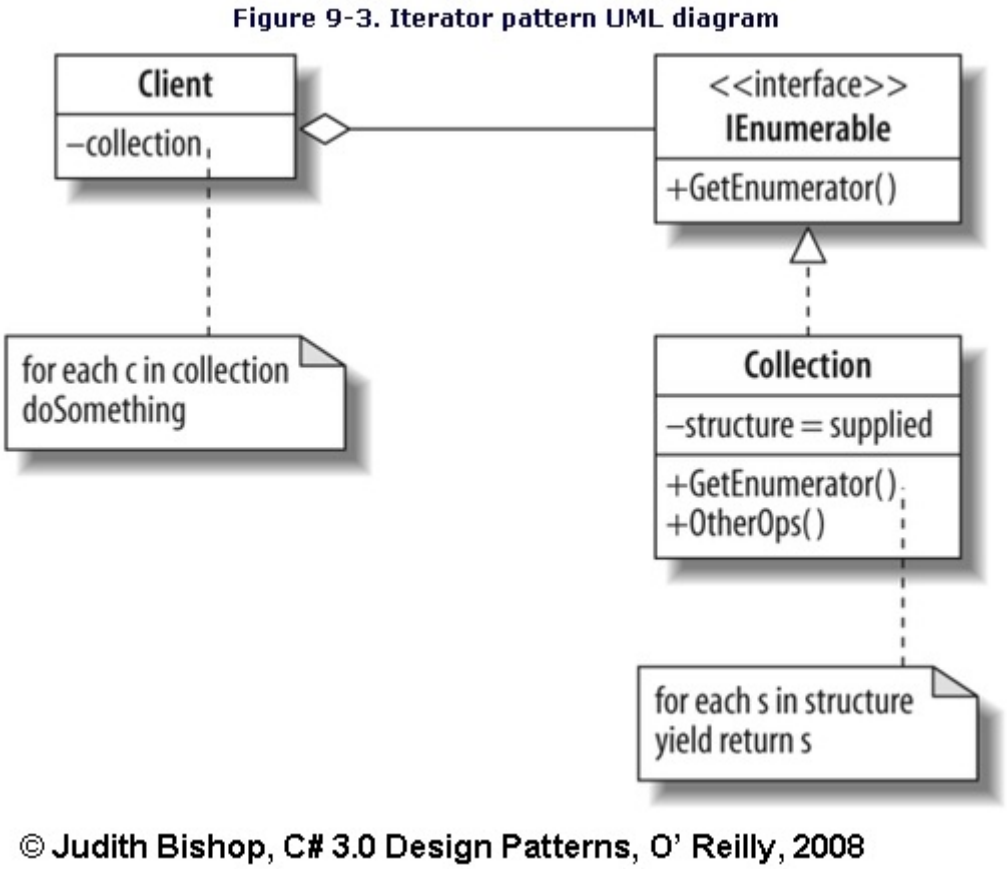
Not thread safe unless its a robust iterator that allows insertions and deletions. This can be be solved by letting the Iterator use a Memento to capture the state of an iteration.

Iterator pattern is not only about traversing through a collection, we can provide different kind of iterators based on our requirements. Iterator pattern hides the actual implementation of traversal through the collection and client programs just use iterator methods.

Let's understand this pattern with a simple example. Suppose we have a list of Radio channels and the client program want to traverse through them one by one or based on the type of channel, for example some client programs are only interested in English channels and want to process only them, they don't want to process other types of channels.

So we can provide a collection of channels to the client and let them write the logic to traverse through the channels and decide whether to process them. But this solution has lots of issues such as client has to come up with the logic for traversal. We can't make sure that client logic is correct and if the number of client grows then it will become very hard to maintain.

Structure



Sample Code

```
```

Memento Pattern

The memento pattern is a close relative of the command pattern, with the important difference that it is used to capture the complete state of an object so that it can be subsequently reset.

The memento pattern captures the complete state of an object into a memento that can be used to reset the object at a later date.

The memento pattern allows a complete reset of an object without the need to track and apply individual undo commands.

Use this pattern when there is a “known-good” point in an object’s life that you may want to return to at some point in the future.

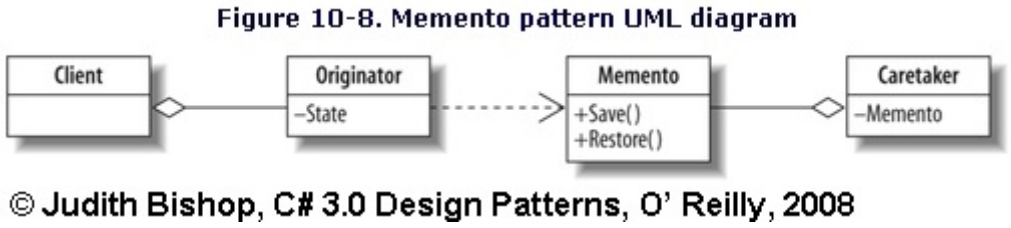
This pattern should be used only when you need to return an object to an earlier state. Use the command pattern, if you need to add support for undoing the effect of only the most recent operation.

The pattern is implemented correctly if the object can be returned to an earlier state from any starting position.

The most common pitfall is to not completely capture or set the state.

The memento and command patterns share a common philosophy.

Structure



Sample Code

```
```

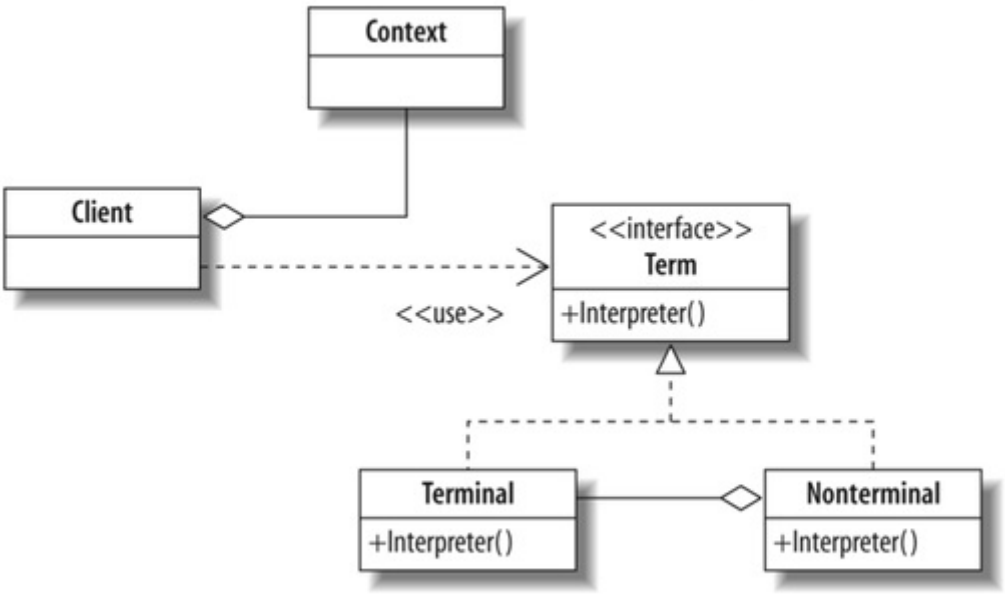
Interpreter Pattern

Is used to define a grammatical representation for a language and provides an interpreter to deal with this grammar. The best example of this pattern is java compiler that interprets the java source code into byte code that is understandable by JVM. Google Translator is also an example of interpreter pattern where the input can be in any language and we can get the output interpreted in another language.

To implement interpreter pattern, we need to create Interpreter context engine that will do the interpretation work and then we need to create different Expression implementations that will consume the functionalities provided by the interpreter context. Finally we need to create the client that will take the input from user and decide which Expression to use and then generate output for the user.

Structure

Figure 10-5. Interpreter pattern UML diagram



Sample Code

Software Refactoring

Chapter Exercises

1. Create Test cases to test All the Creational Design Patterns, The Structural Adaptor Design Pattern(Both Object and Class) and the behavioural Chain of Responsibility Pattern
2. The code for the rest of the Design Patterns has been deliberately left out in this Chapter. Visit the web site http://www.avajava.com/tutorials/categories/design-patterns

and study the code samples for the rest of the design patterns in this chapter. Using the knowledge from the above website, come p with your own new situations and show how you can apply the design patterns for each pattern without sample code. All creational Patterns have been created based on samples from the site. Write your own sample code with test cases.

Chapter 6: Domain Driven Design

Chapter Objectives

1. Understand what Domain Driven Design is and when and why it is valuable to software intensive organizations.
2. Describe how the principle of ” separation of concerns ” has been applied to the main system design
3. Know the the basic principles and processes needed to develop the useful sort of models, tie them into implementation and business analysis, and place them within a viable, realistic strategy.
4. Context Mapping: A pragmatic approach to dealing with the diversity models and processes on real large projects with multi-team/multi-subsystem development.
5. Combining the Core Domain and Context Map to illuminate Strategic Design options for a project.

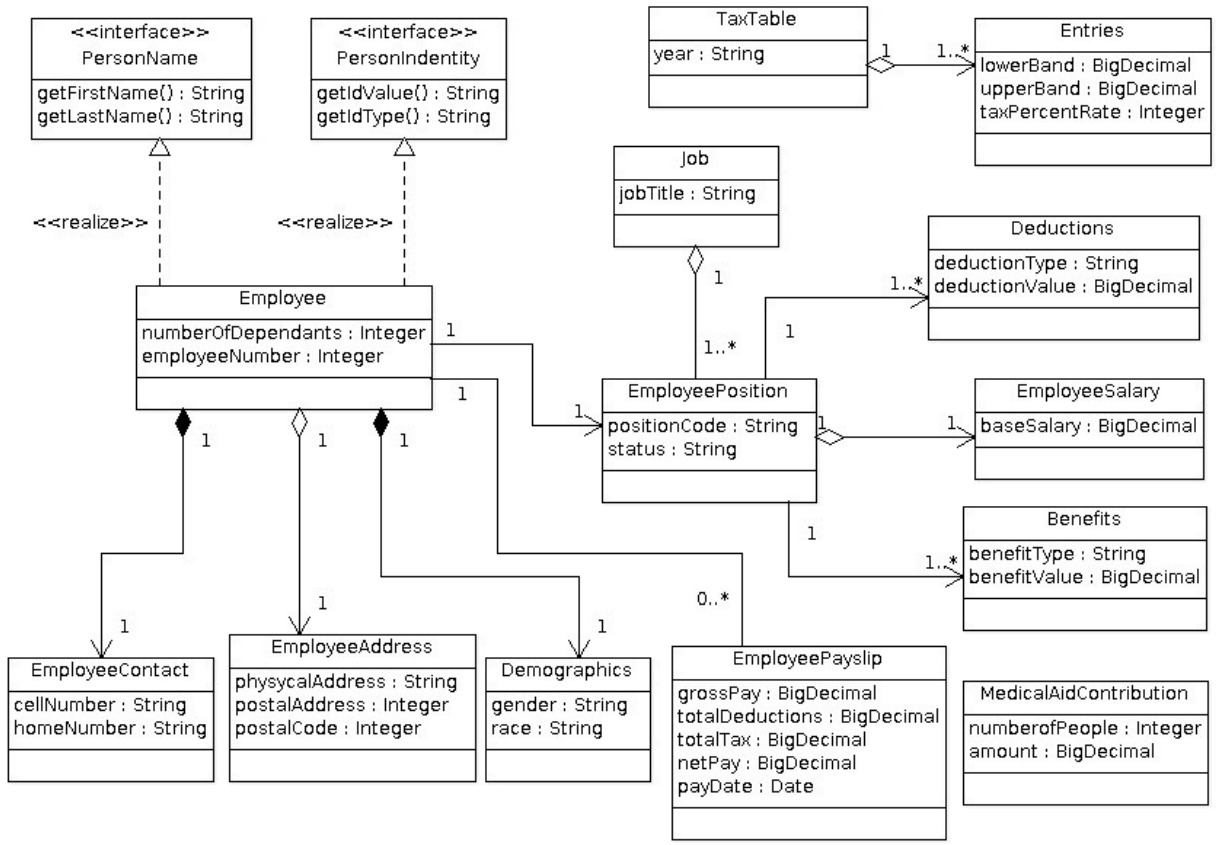
Introduction

Domain Driven Design (DDD) is an approach of how to model the core logic of an application. The term itself was coined by Eric Evans in his book ”Domain Driven Design”. The basic idea is that the design of your software should directly reflect the Domain and the Domain-Logic of the (business-) problem you want to solve with your application. That helps understanding the problem as well as the implementation and increases maintainability of the software. For any application that you develop, the core concept is the domain model, which is the actual representation of how the business works.

The Domain Driven Design approach introduces common principles and patterns that should be used when modelling your Domain. There are the ”building blocks” that should be used to build your domain model and principles that helps to have a nice ”supple design” in your implementation.

Domain-driven design flows from the premise that the heart of software development is knowledge of the subject matter and finding useful ways of understanding that subject matter. The complexity that we should be tackling is the complexity of the domain itself– not the technical architecture, not the user interface, not even specific features. This means designing everything around our understanding and conception of the most es- sential concepts of the business and justifying any other development by how it supports that core.

Domain Model



The word domain her means the area of interest in the business. When you are developing a system to automate activities, you are actually modeling that business. The abstractions that you design, the behaviors that you implement, the UI interactions that you build, all reflect the business – together they constitute the model of the domain.

More formally, a domain model is a blueprint of the relationships between the various entities of the problem domain and sketches out other important details, such as

- Objects that belong to the domain. E.g. for the banking domain, you have objects like bank, account, transactions etc.
- Behaviors that those objects demonstrate in interacting amongst themselves. E.g. in a banking system you debit from an account, you issue a statement to your client. These are some typical interactions that occur between the objects of your domain.
- he language that the domain speaks. When you are modeling the domain of personal banking, terms like debit, credit, portfolio etc. or phrases like “transfer R 100 from account1 to account2” occur quite ubiquitously and form the vocabulary of the domain.
- The context within which the model operates, which includes the set of assumptions and constraints which are relevant to the problem domain and are automatically applicable for the software model that you develop. A new bank account can be opened for a living person or entity only – this can be one of the assumptions that define a context of our domain model for personal banking.

Challenges

Like any other modeling exercise, the most challenging aspect of implementing a domain model is to manage its complexity. Some of these complexities are inherent to the problem, you really can't avoid them. These are called the essential complexities of the system. E.g. when you apply for a personal loan in your bank, determining the eligibility of amount depending on your profile has a fixed complexity that is determined by the core business rules of the domain. This is an essential complexity that you cannot avoid in your solution model.

But some complexities are often introduced by the solution itself, e.g. we implement a new banking solution that introduces extraneous load on operations in the form of additional batch processing. These are known as the incidental complexities of the model.

One of the essential ideas of an efficient model implementation is to reduce the amount of incidental complexity. And more often than not, we find that you can reduce the incidental complexities of a model by adopting techniques that help you manage complexities better.

For example, if your technique leads to better modularization of your model, then your final implementation is not a single monolithic unmanageable piece of software. It's actually decomposed into multiple smaller components, each of which functions within its own context and assumptions. With a modular system each component is self-contained in functionality. And interacts with other components only through explicitly defined contracts, which we call the context boundaries. This helps manage complexity better than a monolithic system.

The Bounded Context

In section previous section we talked about modular models and discussed a few advantages that modularization brings to a domain model. In the world of domain driven design, we use the term bounded context to denote one such module within our model. When we consider a banking system, for example, a portfolio management system, tax and regulatory reports, and term deposit management can be designed as separate bounded contexts. A bounded context is typically at a fairly high level of granularity and denotes one complete functionality within your system.

But when you have multiple bounded contexts in your complete domain model, how do you communicate between them? Remember, each bounded context is self-contained as a module but can have interactions with other bounded contexts. Typically when you design your model, these communications are implemented as explicitly specified set of services or interfaces.

The basic idea is to keep these interactions to the bare minimum so that each bounded context is cohesive enough within itself and yet loosely coupled with other bounded contexts. While bounded contexts define the boundaries of individual components within your model, we'll look at what's within each bounded context in the next section. These are some of the fundamental domain modeling elements that make up the guts of your model.

Model Elements

Various kinds of abstractions define your domain model. If someone asks you to list a few elements from the domain of personal banking, chances are high that you will be naming items like bank, account, account types like checking, savings, money market, transaction types like debit, credit etc. But you will soon realize that many of these elements are similar with respect to how they are created, processed through the pipeline of the business and ultimately evicted from the system. As an example, consider the lifecycle of a client account. Every client account created by the bank passes through a set of states as a result of certain actions from the bank, from the client, or from any other external system.

Entity

Every account has an identity that has to be managed in the course of its entire lifetime within the system. We call such elements with identities as **Entities**. For an account, its identity is its account number. Many of its attributes may change in course of the lifetime of the system. But the account will always be identified with the specific account number that was allocated to it when it was opened.

Two accounts in the same name and having same attributes will be considered different entities since the account numbers will differ.

Value Objects

Each account may have an address, the residential address of the account holder. Note an address is uniquely defined by the value that it contains. You change any attribute of an address and it becomes a different address. An address doesn't have any identity – it's identified entirely based on the value it contains. So we call such objects **Value Objects**.

Diffrences between Entities and Value Objects

So one other way to distinguish between entities and value objects is by the fact that value objects are immutable – you cannot change the contents of a value object without changing the object itself, once you create it.

The difference between an entity and a value object is one of the most fundamental concepts in domain modeling and you must ensure that you have a clear understanding of this. When we talk of an account, we talk of a specific instance of account, having an account number, holders' names and other attributes. Some of these attributes combined together form a unique identity of the account. Typically an account number is this identifying attribute of an account. So even if you have 2 accounts that have the same values for the non- identifying attributes (e.g. holder's name, date of opening etc.), they are 2 different accounts if the account numbers are different.

An account is an entity that has a specific identity. But an address is something of which we need to consider only the value part. So within the model we can choose to have only one instance of a particular address and share it across all accounts that belong to the holders residing at that address. It's only the value that matters. You can change some of the attribute values in an entity and yet the identity doesn't change – e.g. you can change the address of an account and yet it points to the same account. But you cannot change the value of a value object; otherwise it will be a different value object. So a value object is immutable by definition.

Services

The heart of any domain model is the set of behaviors or interactions between the various domain elements. These

are behaviors that are at a higher level of granularity than individual entities or value objects. In fact we consider these as the principal services that the model offers.

As an example from the banking system, a customer comes to the bank or the ATM and transfers money between 2 accounts. This results in a debit from one account, credit to another, which will reflect as a change in balance in the respective accounts. There are some validation checks to be done, whether the accounts are active or not, whether the source account has enough funds to transfer and so on. Note in every such interaction you may have many domain elements involved, both entities and value objects.

In domain driven design we model this entire set of behaviors as one or more Services. Depending on the architecture and the specific bounded context of the model, you can either package it as a stand-alone service (we can name it TransferService), or as part of a collection of services in a more generic module named BankingService. The main point where a domain service differs from an entity or a value object is the level of granularity. In a service, multiple domain entities interact according to specific business rules and deliver a specific functionality in the system. A service is a more macro level element than an entity or a value object. It encapsulates a complete business operation that has a certain value to the user or the bank.

Lifecycle of a Domain Object

Every object (entity or value object) that you have in any model must have a definite lifecycle pattern. What this means is that for every type of object you have in your model, you must have defined ways to handle each of the following events:

- Creation – how the object is created within the system. In the banking system you may have some special abstraction that is responsible for creation of every bank account.
- Participation in behaviors – how the object is represented in memory when it interacts within the system. This is simply the way you model an entity or a value object within your system. Note that a complex entity may consist of other entities as well as value objects. As an example, an account entity may have references to other entities like bank or other value objects like address or account type.
- Persistence – how the object is maintained in the persistent form. This includes issues like how you write the element to the persistent storage and how you retrieve the details in response to queries by the system. If your persistent form is the relational database, then how do you insert, update, delete or query an entity like account.

As always, a uniform vocabulary helps. In the following we will use specific terminologies to refer to how we handle each of the above three lifecycle events in our model. We will call them patterns since we will be using them repeatedly in various contexts of our domain modeling

Factories

It is always a good practice to have dedicated abstractions that handle various parts of the your dmain lifecycle. Instead of littering the entire code base with snippets of the code that creates your entities, centralize them using a pattern. This serves two purposes:

- It keeps all creational code in one place
- It abstracts the process of creation of an entity from the caller

As an example, you can have an account factory that takes the various parameters needed to create an account and hands you over a newly created account. The new account that you get back from the factory may be a checking, savings or money market account depending on the parameters you pass. So the factory lets you create different types of objects using the same API. It abstracts the process and the actual type of created objects.

The creation logic resides within a factory. But where does the factory belong? A factory after all provides you a service – the service of creation and possible initialization. It's the responsibility of the factory to hand you over a fully constructed minimally valid instance of the domain object. One option is to make the factory part of the module that defines the domain object, e.g using the Builder Design Pattern.

Aggregates

An account can be thought of being composed of a group of related objects. Typically this includes core account identifying attributes like account number, various non identifying attributes like holders' names, date when the account was opened, date of closing (if it's a closed account) and reference to other objects like Address, Bank etc.

One way you can visualize this entire graph of objects is to think of it as forming a consistency boundary within itself. By consistency boundary we mean that when we have an account instantiated, all of these individual participating objects and attributes must be consistent as per the business rules of the domain.

We cannot have an account with date of closing preceding the date of opening. Or we cannot have an account without any holders' names in it. These are all valid business rules and the instantiated account must have all composing objects honor each of these rules. Once we identify this set of participating objects in the graph, we call this graph an Aggregate.

An aggregate can consist of one or more entities and value objects (and of course other primitive attributes). Besides ensuring the consistency of business rules, an aggregate within a bounded context is also often looked upon as a transaction boundary in the model. One of the entities within the aggregate forms the aggregate root – it's sort of the guardian of the entire graph and serves as the single point of interaction of the aggregate with its clients. So the aggregate root has 2 objectives to enforce:

- Ensure consistency boundary of business rules and transactions within the aggregate
- Prevent the implementation of the aggregate from leaking out to its clients acting as a façade for all the operations that the aggregate supports.

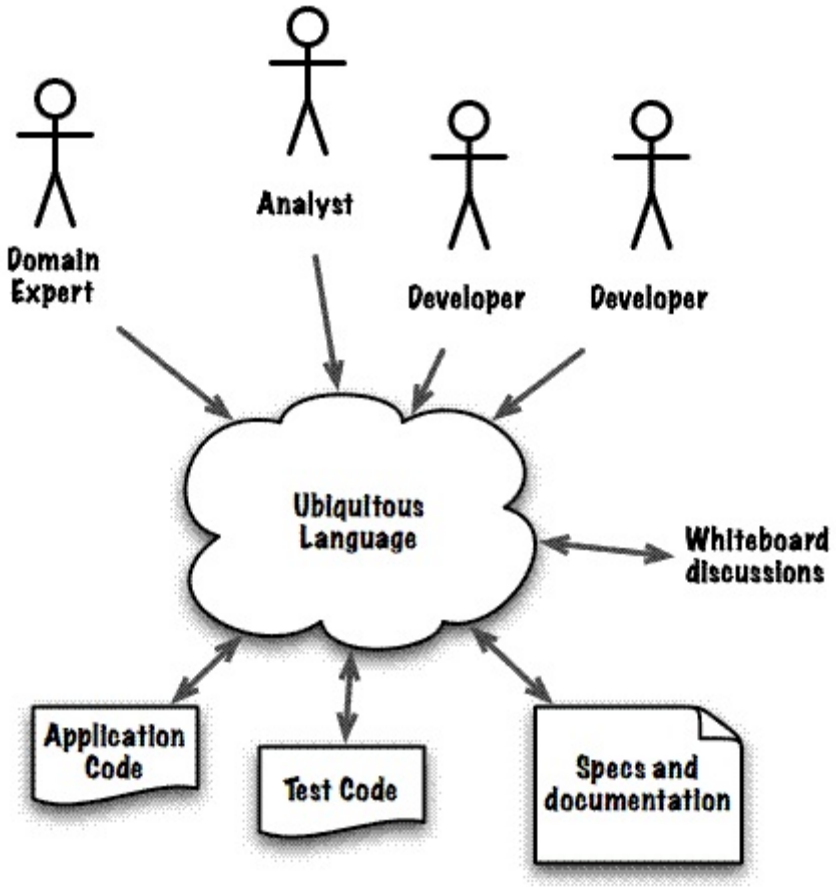
Repositories

As we saw, aggregates are created by factories and represent the underlying entities in memory during the active phase of the objects' lifecycle. But we also need a way to persist an aggregate when we no longer need it. Note we cannot throw it away since we may need to fetch it later for some other purpose.

A Repository gives you this interface for parking an aggregate in some persistent form so that you can fetch it back to an in memory entity representation when you need it. Usually a repository has an implementation based on some persistent storage like an RDBMS, though the contract doesn't enforce that too . Also note that the persistent model of the aggregate may be entirely different from the in-memory aggregate representation and is mostly driven by the underlying storage data model. It's the responsibility of the repository to provide you the interface to manipulate entities from the persistent storage without exposing the underlying relational (or whatever model the underlying storage supports) data model.

Note that the interface for a repository doesn't have any knowledge about the nature of the underlying persistent store. It can be a relational database or a NoSQL database – only the implementation knows that. So what an aggregate offers for in memory representation of the entity, a repository does the same for the persistent storage. An aggregate hides the underlying details of the in-memory representation of the object, while a repository abstracts the underlying details of the persistent representation of the object.

The Ubiquitous Language



Now we have the entities, value objects and services that form the model, and we know that all these elements need to interact with each other to implement the various behaviors that the business executes. As a software craftsman it's your responsibility to model this interaction in such a way that it's understandable not only to the hardware underneath but also to a curious human mind. This interaction needs to reflect the underlying business semantics and must contain vocabularies from the problem domain you are modeling.

By vocabulary we mean the names of participating objects and the behaviors that get executed as part of the use cases. In our example, entities like bank, account, customer and balance, and behaviors like debit and credit, resonate strongly with the terminologies of the actual business and hence form part of the domain vocabulary. Not only these stand-alone elements, use of domain vocabulary needs to be transitively extended to larger abstractions formed out of smaller ones. Make sure all the players involved understand the terms used.

Use the domain vocabulary in your model and make the terms interact in such a way that it resembles the language that the domain speaks. Start with the correct naming of entities and atomic behaviors and extend this vocabulary to larger abstractions that you compose out of them. Different modules can speak different dialects of the language and the same term may mean something different in a different bounded context. But within the context the vocabulary should be clear and unambiguous.

Having a consistent ubiquitous language has a lot to do with designing proper APIs of your model. The APIs must be expressive so that a person who is expert in the domain can understand the context looking at the API only.

Chapter Exercises

Model and Project

Create a github project based on the APPROVED project from ME. This will be your individual project for the rest of your third year. Your project should have the following

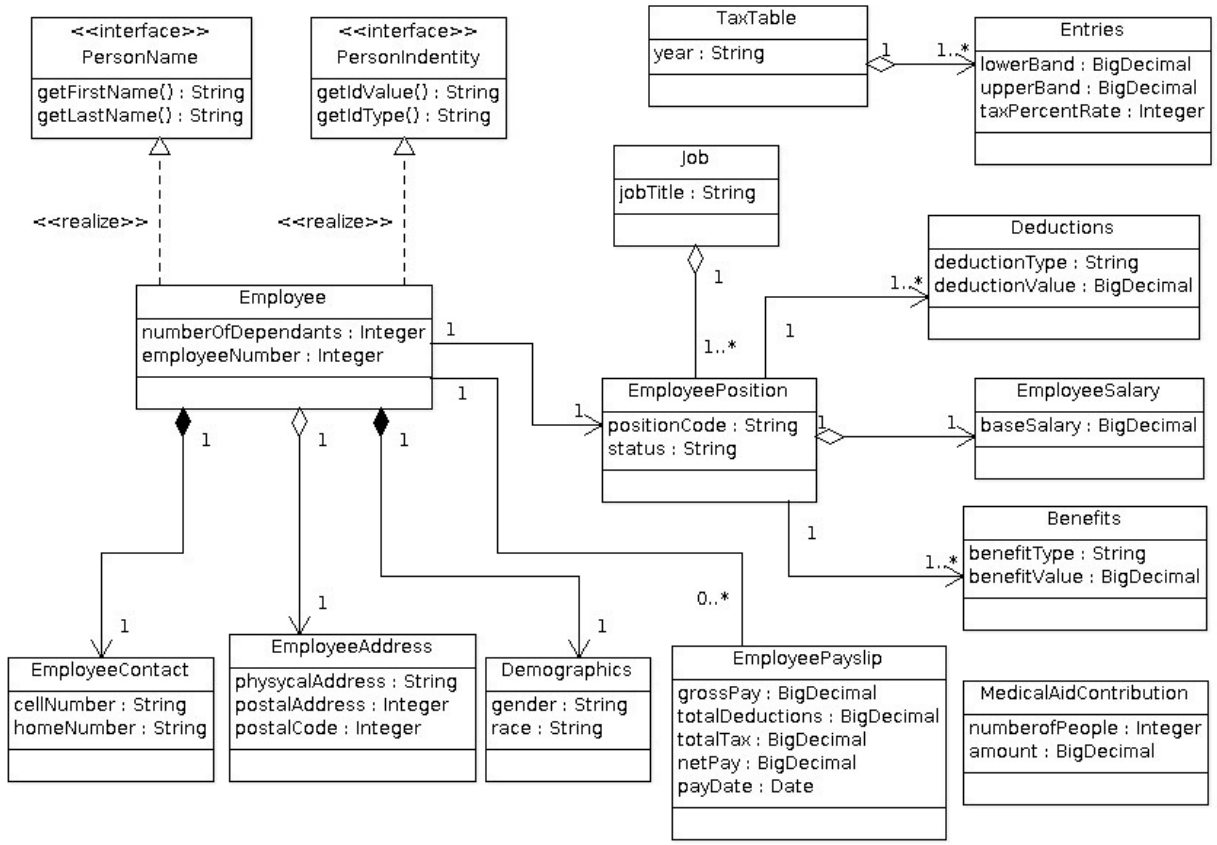
1) Basic Code from the Quick Start Maven Archetype(You don't have to add any Spriing Boot Jars or Database Connectivity Jars, just the quick start project created initially)

2) Your Project should have a README.md. In your README file you should have the project title, background and description of the problem you are trying to solve

3) A domain model of the project with AT LEAST 10 classes with their respective relations ships in UML

Please Submit the link to your GitHub Project. Please DO NO SUBMIT your ZIPPED PROJECT.

See the UML diagram below for the example of the Narration below



According to the Model, the employee, with a mandatory employee number, has names, demographic information, contact and Address. The company also keeps track of the number of dependents because they need this information to calculate the amount of money they contribute towards an employee's medical monthly contribution as a benefit.

The company also keeps track of all the types of identities and values that an employee wish to use. For example, the employee can have a type of identity called Passport with a value of “ZG1234”.

The Domain Model also shows a Tax Entry Table used to calculate the tax rate paid by an employee based on a particular year. For example an employee whose Gross Pay falls between the Lower band of R10 000 and Upper band of R 20 000 will be taxed at a rate of 20 % and the amount value will be used as a tax deduction. See Tax table on next page for details.

Also shown in the Domain Model is the Job Entity with a title. Each Job can have several Positions uniquely identified with a position code. An employee can only fill one position at a time. If the position is filled, the status is changed to CLOSED, otherwise it is left as OPEN. Attached to a position is the base salary for that position together with benefits and deductions. Deductions comprise amounts like tax and loan repayments. Benefits includes amounts for medical Aid contributions, allowances and bonuses.

Every month the company generates a Payslip for each employee, showing the Gross Pay, Total Deductions, Total Tax, Net pay and the Pay Date.

Model Factories

Create factories and provide Tests for your factory lifecycle. Your Cycle should include creation and update of entities or value objects.

Repositories

Create repository interface for your entities and test for CREATE, READ , UPDATE and DELETE. Make Sure u write all Test Cases.

Chapter 7: Microservice Architectures

Chapter 8: Application Security

Chapter 9: Front end Mobile Platform Developmen

Chapter 10: Communication Services Mobile Platform Development