

Programming Style Guides and Coding Standards

- A programming style guide is an opinionated guide of programming conventions, style, and best practices for a team or project.
- A team following a style guide helps everyone write code in a consistent way, and consistent code is easier to read and faster to update.
- Consistent code is easier to read and understand making it faster to add new features.

Widely accepted style guides that you should consider to start with:

- Google's Style Guides
- JavaScript Standard Style
- GitHub's Ruby Style Guide
- Python Foundation's Style Guide
- Airbnb's JavaScript Style Guide
- Angular's Style Guide

- Implementing one of these generally accepted style guides is a good start to helping your team write code consistently.
- Make your style guide easy to reference.
- Style guide acts as a basic code blueprint for your team during all parts of the software development lifecycle.

- To keep relevant, it should be often discussed. Such as during a developers onboarding, when writing code, when writing tests, and during code review.

Literature Programming

- It is a methodology that combines a programming language with a documentation language, thereby making programs more robust, more portable, more easily maintained, and arguably more fun to write than programs that are written only in a high-level language.

- The main idea is to treat a program as a piece of literature, addressed to human beings rather than to a computer.
- The program is also viewed as a hypertext document, rather like the World Wide Web.

Software documentation

- Software documentation is a part of any software. Appropriate details and description need to be in the documented to achieve the following goals:
 - Resolve issue encountered by the developer during the development process
 - Help end-user to understand the product
 - Assist customers and the support team to find the information.

Javadoc

- JavaDoc tool is a document generator tool in Java programming language for generating standard documentation in HTML format.
- It generates API documentation.

- Before using JavaDoc tool, you must include JavaDoc comments
`/** */` providing information about classes, methods, and constructors, etc.
- For creating a good and understandable document API for any java file you must write better comments for every class, method, constructor.

JavaDoc Tool

```
/**  
 * JavaDoc comment  
 */
```

@author

@param

@see

@version

@return

- The JavaDoc comments is different from the normal comments because of the extra asterisk at the beginning of the comment. It may contain the **HTML tags** as well.

// Single-Line Comment

/*

Multiple-Line comment

*/

/**

JavaDoc comment

*/

- By writing a number of comments, it **does not affect the performance** of the Java program as all the comments are removed at compile time.

Generation of JavaDoc

To create a JavaDoc you do not need to compile the java file. To create the Java documentation API, you need to write Javadoc followed by file name.

javadoc file_name or javadoc package_name

After successful execution of the above command, a number of HTML files will be created, open the file named index to see all the information about classes.

Tag	Parameter	Description
-----	-----------	-------------

@author	author_name	Describes an author
---------	-------------	---------------------

@param	description	provide information about method parameter or the input it takes
--------	-------------	--

@see	reference	generate a link to other element of the document
------	-----------	--

@version	version-name	provide version of the class, interface or enum.
----------	--------------	--

@return	description	provide the return value
---------	-------------	--------------------------

phpDocumentor

- phpDocumentor is an application that is capable of analyzing your PHP source code and DocBlock comments to generate a complete set of API Documentation.

phpDocumentor v3 (Stable)

- v3 is the latest stable release.
- The easiest way to run phpDocumentor is by running the following command:

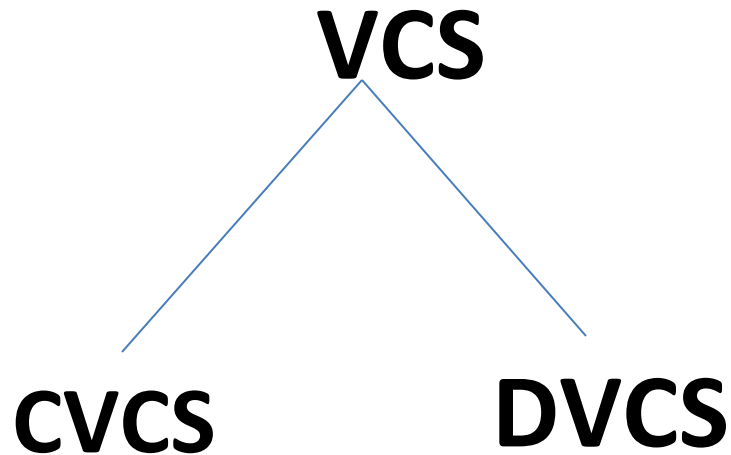
```
$ phpdoc run -d <SOURCE_DIRECTORY> -t  
    <TARGET_DIRECTORY>
```


**VERSION CONTROL
SYSTEMS BASIC
CONCEPTS**

Version Control System

- **Version Control System (VCS)** is a software that helps software developers to work together and maintain a complete history of their work.
- Listed below are the functions of a VCS –
 - Allows developers to work simultaneously.
 - Does not allow overwriting each other's changes.
 - Maintains a history of every version.

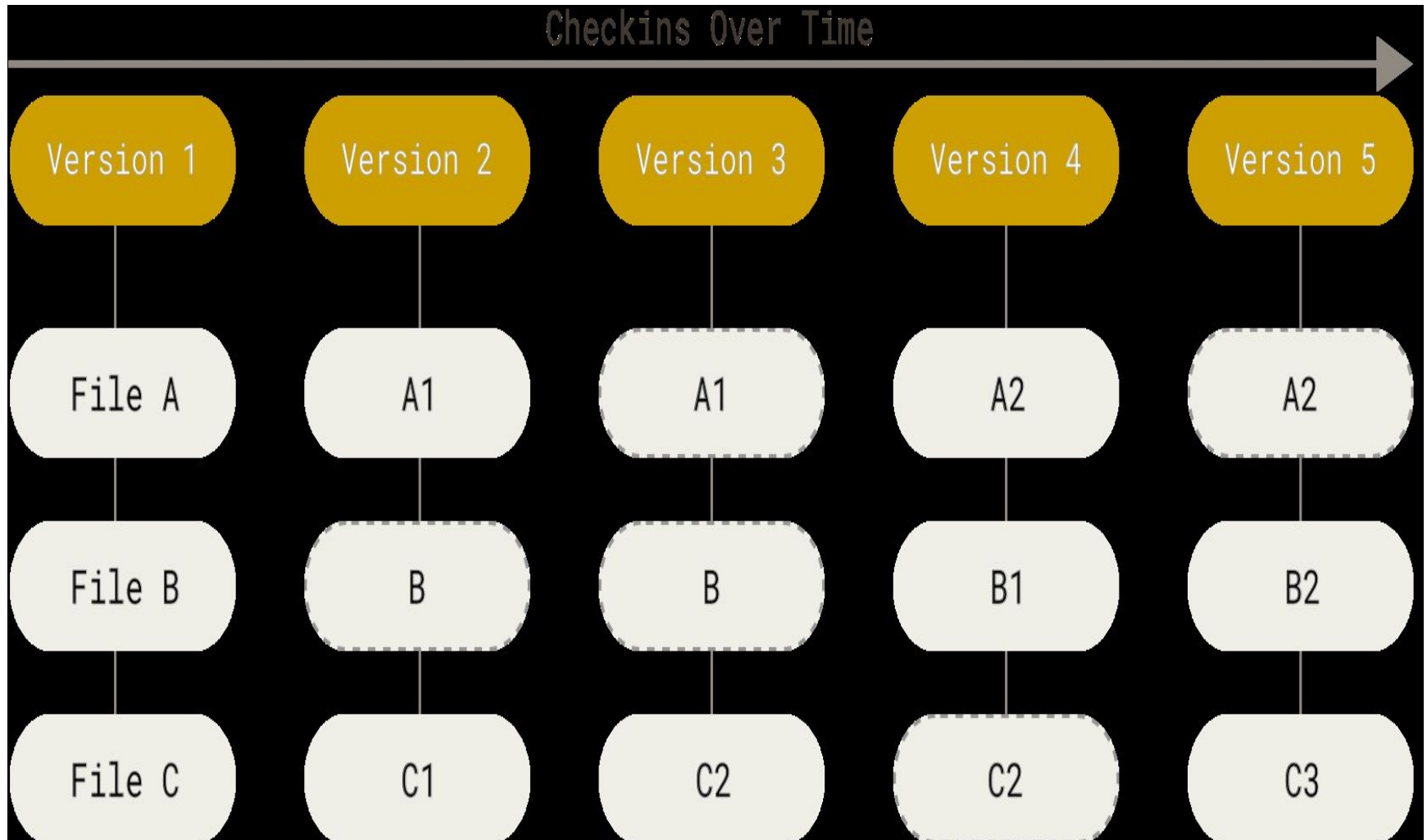
- Following are the types of VCS –
 - Centralized version control system (CVCS).
 - Distributed/Decentralized version control system (DVCS).



- Linux development developed their own tool. Some of the goals of the new system were as follows:
 - Speed
 - Simple design
 - Strong support for non-linear development
 - Fully distributed
 - Able to handle large projects like the Linux kernel efficiently

- Git thinks of its data more like a series of snapshots of a miniature file system.
- Every time when committing, or saving the state of project, Git basically takes a picture of what all 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.
- Git thinks about its data more like a **stream of snapshots**.

Storing data as snapshots of the project over time



Git Has Integrity

- Everything in Git is check summed before it is stored and is then referred to by that checksum.
- It's impossible to change the contents of any file or directory without Git knowing about it.
- This functionality is built into Git at the lowest levels and is integral to its philosophy.
- Information is not lost in transit or get file corruption without Git being able to detect it.

- The mechanism that Git uses for this checksumming is called a SHA-1 hash.
- This is a 40-character string composed of hexadecimal characters (0–9 and a–f) and calculated based on the contents of a file or directory structure in Git.
- Git stores everything in its database not by file name but by the hash value of its contents.
- A SHA-1 hash looks something like this:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```


Git Generally Only Adds Data

- When action is done in Git, it only *add data to the Git database. It is hard to get the system to do anything that is not undoable or to make it erase data in any way.*
- But after committing a snapshot into Git, it is very difficult to lose, especially if database is pushed to another repository.

The Three States

- Git has three main states that your files can reside in: ***modified, staged, and committed:***
- Modified means that you have changed the file but have not committed it to your database yet.
- Staged means that you have marked a modified file in its current version to go into your next commit snapshot.
- Committed means that the data is safely stored in your local database.

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

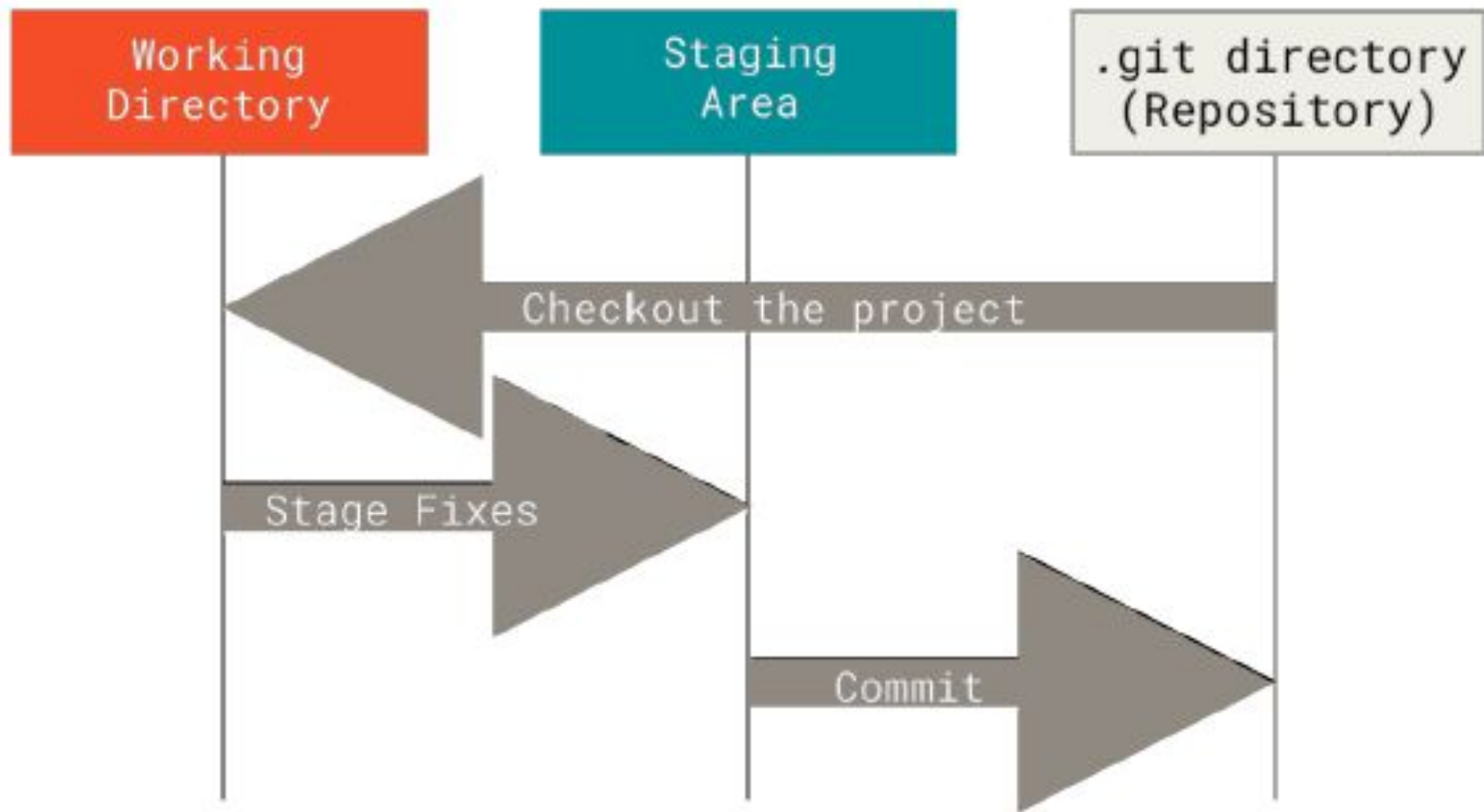


Figure 6. Working tree, staging area, and Git directory

- The working tree is a single checkout of one version of the project. These 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. Its technical name in Git parlance is the “index”, but the phrase “staging area” works just as well.

- The Git directory is where Git stores the metadata and object database for your project. This is the most important part of Git, and it is what is copied when you *clone a repository from another* computer.
- If a particular version of a file is in the Git directory, it's considered *committed*.
- If it has been modified and was added to the staging area, it is *staged*.
- *If it was changed since it was checked out* but has not been staged, it is *modified*.

SETTING UP GIT

- Git has a tool called git config that get and set configuration variables that control all aspects of how Git looks and operates.
- These variables can be stored in three different places:
 1. [path]/etc/gitconfig file
 2. ~/.gitconfig or ~/.config/git/config file
 3. config file in the Git directory

1. [path]/etc/gitconfig file:
 - It contains values applied to every user on the system and all their repositories.
 - If the option `–system` is passed to `git config`, it reads and writes from this file specifically.
 - Because this is a system configuration file, it needs administrative or superuser privilege to make changes to it.

2. `~/.gitconfig` or `~/.config/git/config` file:

- Values specific personally to the user.
- Git can read and write to this file specifically by passing the `--global` option, and this affects *all of the repositories you work with on your system*.

3. config file in the Git directory

- It is specific to that single repository.
- Git can be forced to read from and write to this file with the `--local` option, but that is in fact the default.
- You need to be located somewhere in a Git repository for this option to work properly.

- On Windows systems, Git looks for the .gitconfig file in the \$HOME directory (C:\Users\%USER%).
- There is also a system-level config file at C:\Documents and Settings\All Users\Application Data\Git\config on Windows XP, and in
- C:\ProgramData\Git\config on Windows Vista and newer.
- This config file can only be changed by git config -f <file> as an admin.

```
$ git config --list --show-origin
```

1. After installing Git, set your user name and email address.

```
$ git config --global user.name "John Doe"  
$ git config --global user.email johndoe@example.com
```

2. Configure the default text editor that will be used when Git needs you to type in a message. If not configured, Git uses your system's default editor.

If you want to use a different text editor, such as Emacs, you can do the following:

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

- By default Git will create a branch called *master* when you create a new repository with *git init*.
- To set *main* as the default branch name do:

```
$ git config --global init.defaultBranch main
```

The Command Line

- Command line is the only place you can run *all Git commands*.
- By knowing how to run the command-line version, we can probably figure out how to run the GUI version.

Cloning a Git Repository

Git repository can be obtained in one of two ways:

1. You can take a local directory that is currently not under version control, and turn it into a Git repository, or
2. You can *clone an existing Git repository from elsewhere.*

In either case, you end up with a Git repository on your local machine, ready for work.

- Every version of every file for the history of the project is pulled down by default when you run `git clone`.
- In fact, if your server disk gets corrupted, you can often use nearly any of the clones on any client to set the server back to the state it was in when it was cloned.

- You clone a repository with `git clone <url>`.
- For example, if you want to clone the Git linkable library called `libgit2`, you can do so like this:

\$ git clone

<https://github.com/libgit2/libgit2>

- That creates a directory named libgit2, 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 libgit2 directory that was just created, 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 libgit2, you can specify the new directory name as an additional argument:
- **\$ git clone https://github.com/libgit2/libgit2 mylibgit**

VIEWING THE COMMIT HISTORY

- After creating several commits, or if a repository is cloned with an existing commit history and to look back to see what has happened. The most basic and powerful tool to do this is the ***git log*** command.


```
sngist@sngist-PC MINGW32 ~
```

```
$ cd "D:\test"
```

```
sngist@sngist-PC MINGW32 /d/test (main)
```

```
$ git log
```

```
commit 29aec28e752f9a21a055f7ee1ce6455455df00b (HEAD -> main)
```

```
Author: unknown <reashmaraj.p.r@gmail.com>
```

```
Date: Thu Jan 14 12:12:17 2021 +0530
```

```
    first commit
```

```
sngist@sngist-PC MINGW32 /d/test (main)
```

```
$
```



- When you run `git log` in this project, you should get output that looks something like this:

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700
```

Change version number

```
commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700
```

Remove unnecessary test

```
commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700
```

Initial commit

- By default, with no arguments, `git log` lists the commits made in that repository in reverse chronological order; that is, the most recent commits show up first.
- ***git log*** lists each commit with its SHA-1 checksum, the author's name and email, the date written, and the commit message.

- Options `-p` or `--patch` shows the difference (the *patch output*) introduced in each commit.
- The number of log entries displayed can be limited such as using `-2` to show only the last two entries.

```
$ git log -p -2
```

```
commit ca82a6dff817ec66f44342007202690a93763949
```

```
Author: Scott Chacon <schacon@gee-mail.com>
```

```
Date: Mon Mar 17 21:52:11 2008 -0700
```

```
Change version number
```

- To see some abbreviated stats for each commit, you can use the `--stat` option
- The `--stat` option prints below each commit entry a list of modified files, how many files were changed, and how many lines in those files were added and removed. It also puts a summary of the information at the end.


```
sngist@sngist-PC MINGW32 /d/test (main)
```

```
$ git log --stat
```

```
commit 29aec28e752f9a21a055f7ee1ce6455455df00b (HEAD -> main)
```

```
Author: unknown <reashmaraj.p.r@gmail.com>
```

```
Date: Thu Jan 14 12:12:17 2021 +0530
```

```
    first commit
```

```
    README.md | 1 +
```

```
    1 file changed, 1 insertion(+)
```

- The option **--pretty** changes the log output to formats other than the default. The **oneline** value for this option prints each commit on a single line, which is useful if you're looking at a lot of commits.
- The **short, full, and fuller** values show the output in roughly the same format but with less or more information, respectively:


```
sngist@sngist-PC MINGW32 /d/test (main)
```

```
$ git log --pretty=oneline
```

```
29aec28e752f9a21a055f7ee1ce6455455df00b (HEAD -> main) first commit
```

```
sngist@sngist-PC MINGW32 /d/test (main)
```

```
$ git log --pretty=short
```

```
commit 29aec28e752f9a21a055f7ee1ce6455455df00b (HEAD -> main)
```

```
Author: unknown <reashmaraj.p.r@gmail.com>
```

```
first commit
```

```
sngist@sngist-PC MINGW32 /d/test (main)
```

```
$ git log --pretty=full
```

```
commit 29aec28e752f9a21a055f7ee1ce6455455df00b (HEAD -> main)
```

```
Author: unknown <reashmaraj.p.r@gmail.com>
```

```
Commit: unknown <reashmaraj.p.r@gmail.com>
```

```
first commit
```

```
sngist@sngist-PC MINGW32 /d/test (main)
```

```
$ git log --pretty=fuller
```

```
commit 29aec28e752f9a21a055f7ee1ce6455455df00b (HEAD -> main)
```

```
Author: unknown <reashmaraj.p.r@gmail.com>
```

```
AuthorDate: Thu Jan 14 12:12:17 2021 +0530
```

```
Commit: unknown <reashmaraj.p.r@gmail.com>
```

```
CommitDate: Thu Jan 14 12:12:17 2021 +0530
```

```
first commit
```

- The option **format** allows you to specify your own log output format. This is useful when you're generating output for machine parsing because it specifies the format explicitly, you know it won't change with updates to Git:

```
$ git log --pretty=format:"%h - %an, %ar : %s"  
ca82a6d - Scott Chacon, 6 years ago : Change version number  
085bb3b - Scott Chacon, 6 years ago : Remove unnecessary test  
a11bef0 - Scott Chacon, 6 years ago : Initial commit
```

Table 1. Useful specifiers for `git log --pretty=format`

Specifier	Description of Output
<code>%H</code>	Commit hash
<code>%h</code>	Abbreviated commit hash
<code>%T</code>	Tree hash
<code>%t</code>	Abbreviated tree hash
<code>%P</code>	Parent hashes
<code>%p</code>	Abbreviated parent hashes
<code>%an</code>	Author name
<code>%ae</code>	Author email
<code>%ad</code>	Author date (format respects the <code>--date=option</code>)
<code>%ar</code>	Author date, relative
<code>%cn</code>	Committer name
<code>%ce</code>	Committer email
<code>%cd</code>	Committer date
<code>%cr</code>	Committer date, relative
<code>%s</code>	Subject

Table 2. Common options to `git log`

Option	Description
<code>-p</code>	Show the patch introduced with each commit.
<code>--stat</code>	Show statistics for files modified in each commit.
<code>--shortstat</code>	Display only the changed/insertions/deletions line from the <code>--stat</code> command.
<code>--name-only</code>	Show the list of files modified after the commit information.
<code>--name-status</code>	Show the list of files affected with added/modified/deleted information as well.
<code>--abbrev-commit</code>	Show only the first few characters of the SHA-1 checksum instead of all 40.
<code>--relative-date</code>	Display the date in a relative format (for example, “2 weeks ago”) instead of using the full date format.
<code>--graph</code>	Display an ASCII graph of the branch and merge history beside the log output.
<code>--pretty</code>	Show commits in an alternate format. Option values include <code>oneline</code> , <code>short</code> , <code>full</code> , <code>fuller</code> , and <code>format</code> (where you specify your own format).
<code>--oneline</code>	Shorthand for <code>--pretty=oneline --abbrev-commit</code> used together.

- The time-limiting options such as `--since` and `--until` are very useful.
- For example, this command gets the list of commits made in the last two weeks:

```
$ git log --since=2.weeks
```

- The **-S** option takes a string and shows only those commits that changed the number of occurrences of that string.

```
$ git log -S function_name
```

- The last really useful option to pass to git log as a filter is a **path**. If you specify a directory or file name, you can limit the log output to commits that introduced a change to those files.

```
$ git log -- path/to/file
```

Table 3. Options to limit the output of `git log`

Option	Description
<code>-<n></code>	Show only the last n commits
<code>--since, --after</code>	Limit the commits to those made after the specified date.
<code>--until, --before</code>	Limit the commits to those made before the specified date.
<code>--author</code>	Only show commits in which the author entry matches the specified string.
<code>--committer</code>	Only show commits in which the committer entry matches the specified string.
<code>--grep</code>	Only show commits with a commit message containing the string
<code>-S</code>	Only show commits adding or removing code matching the string

GIT BRANCHING

Git Branching

- Branching means you diverge from the main line of development and continue to do work without messing with that main line.
- Git branches are incredibly lightweight, making branching operations nearly instantaneous, and switching back and forth between branches is fast.
- Git encourages workflows that branch and merge often, even multiple times in a day.

- 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 address, 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.

- Let's assume that you have a directory containing three files, and you stage them all and commit.
- Staging the files computes a checksum for each one, stores that version of the file in the Git repository (Git refers to them as *blobs*), and adds that checksum to the staging area:

```
$ git add README test.rb LICENSE
```

```
$ git commit -m 'Initial commit'
```

- Git repository now contains five objects:
 - three *blobs* (each representing the contents of one of the three files)
 - one *tree* that lists the contents of the directory and specifies which file names are stored as which blobs
 - one *commit* with the pointer to that root tree and all the commit metadata.

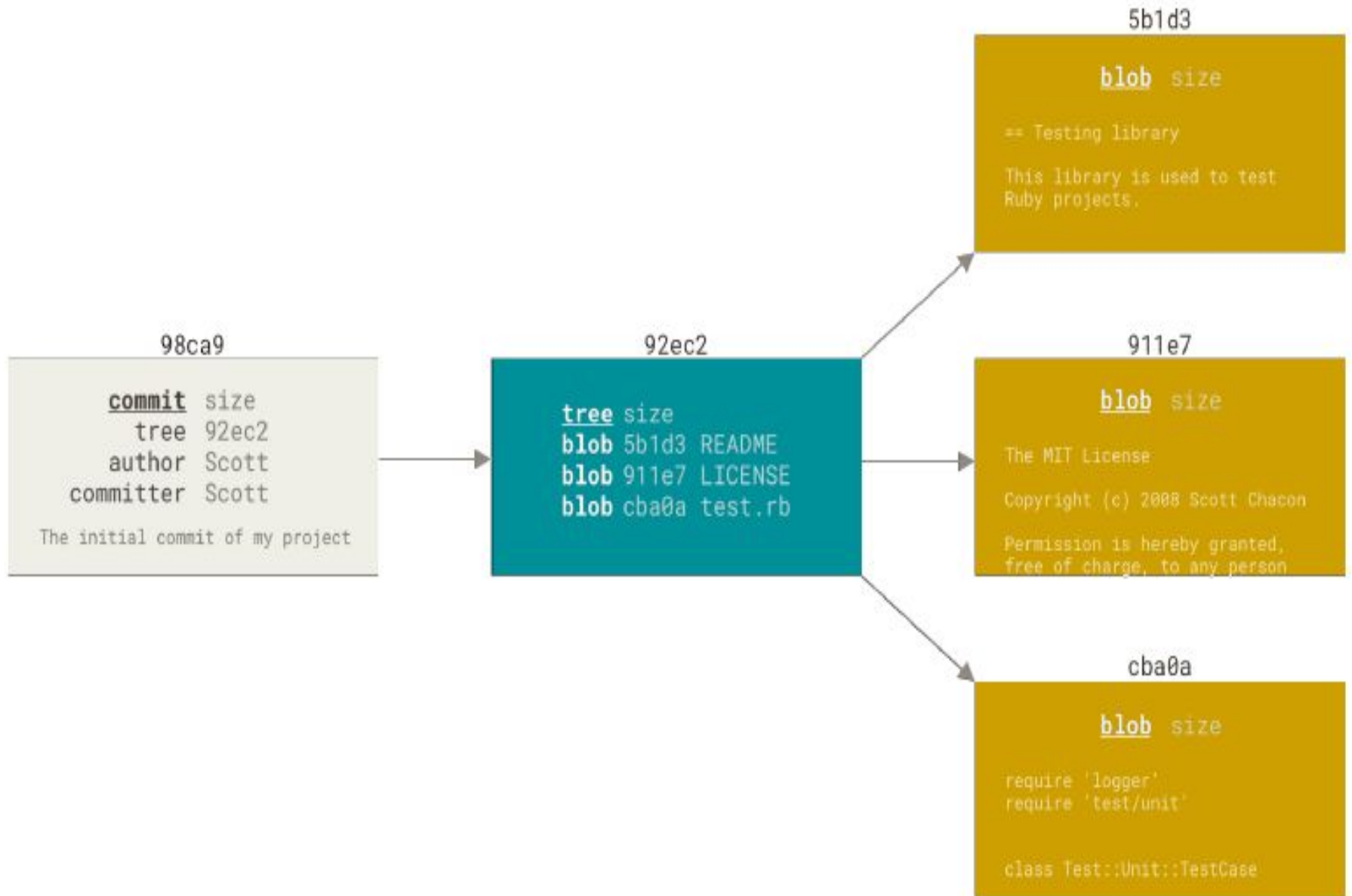


Figure 9. A commit and its tree

If you make some changes and commit again, the next commit stores a pointer to the commit that came immediately before it.

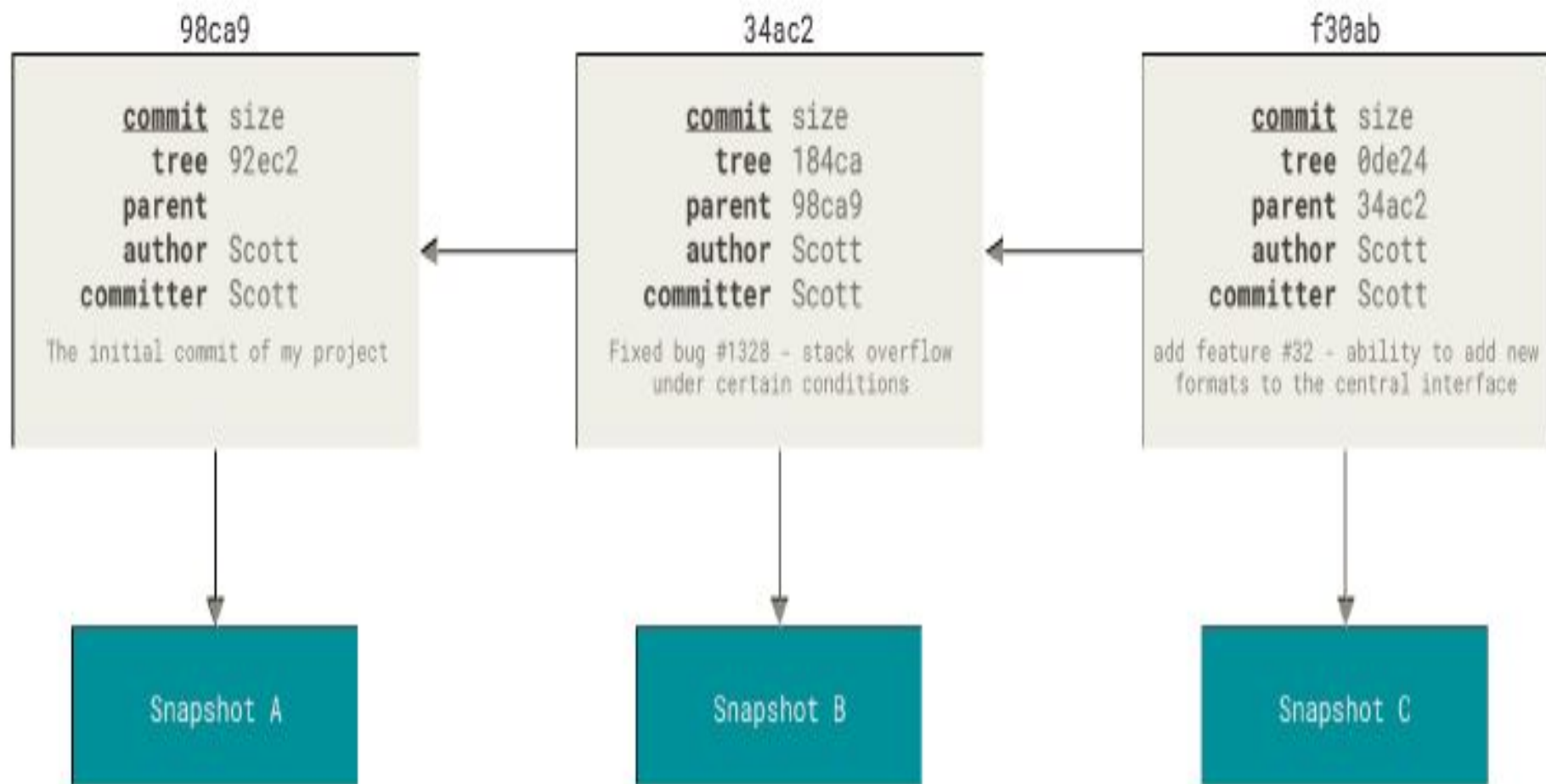


Figure 10. Commits and their parents

- A branch in Git is simply a lightweight movable pointer to one of these commits.
- The default branch name in Git is master.
- As you start making commits, you're given a master branch that points to the last commit you made.
- Every time you commit, the master branch pointer moves forward automatically.

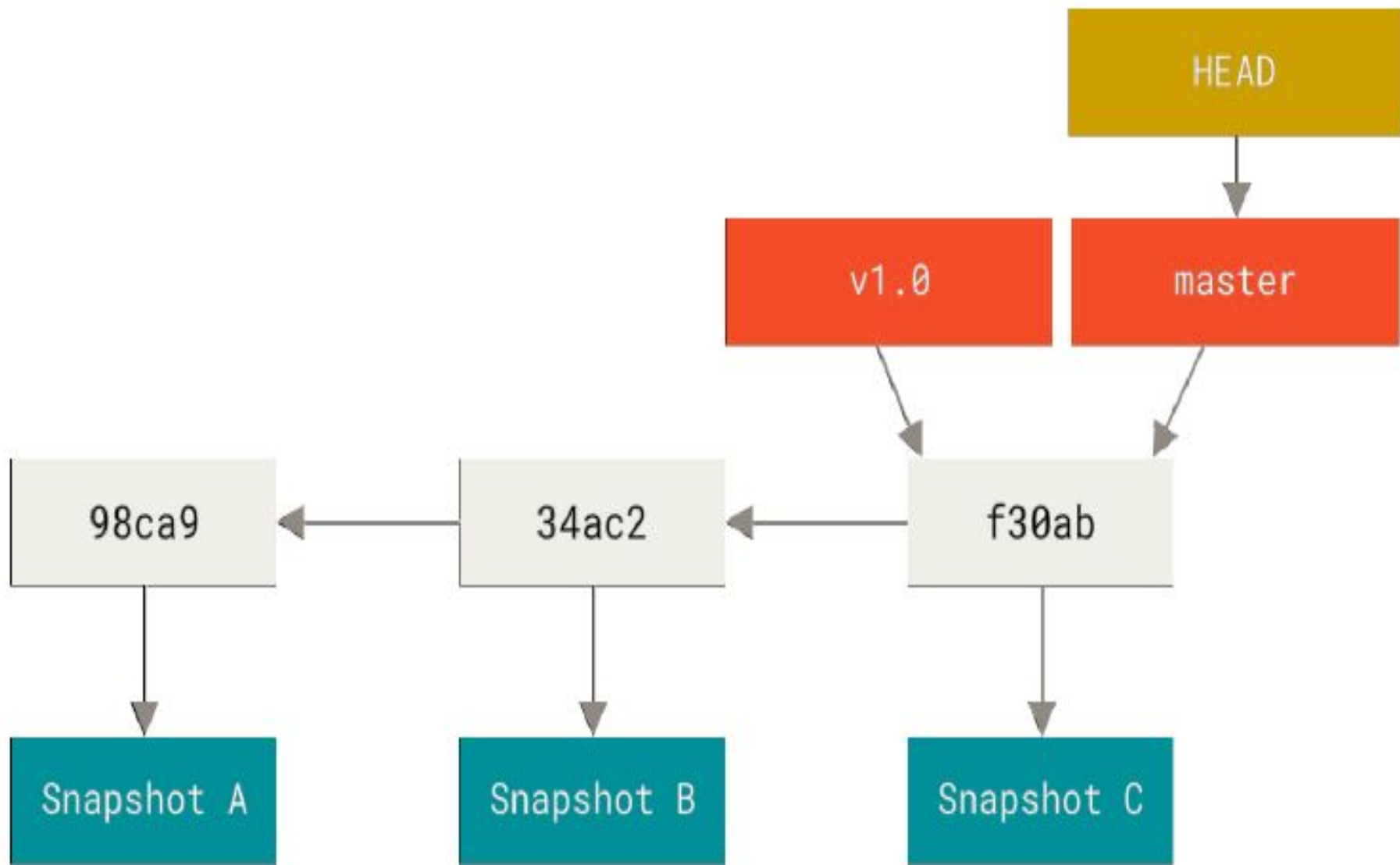


Figure 11. A branch and its commit history

Creating a New Branch

- To create a new branch called testing.
- This is done with the git branch command:

```
$ git branch testing
```

This creates a new pointer to the same commit you're currently on.

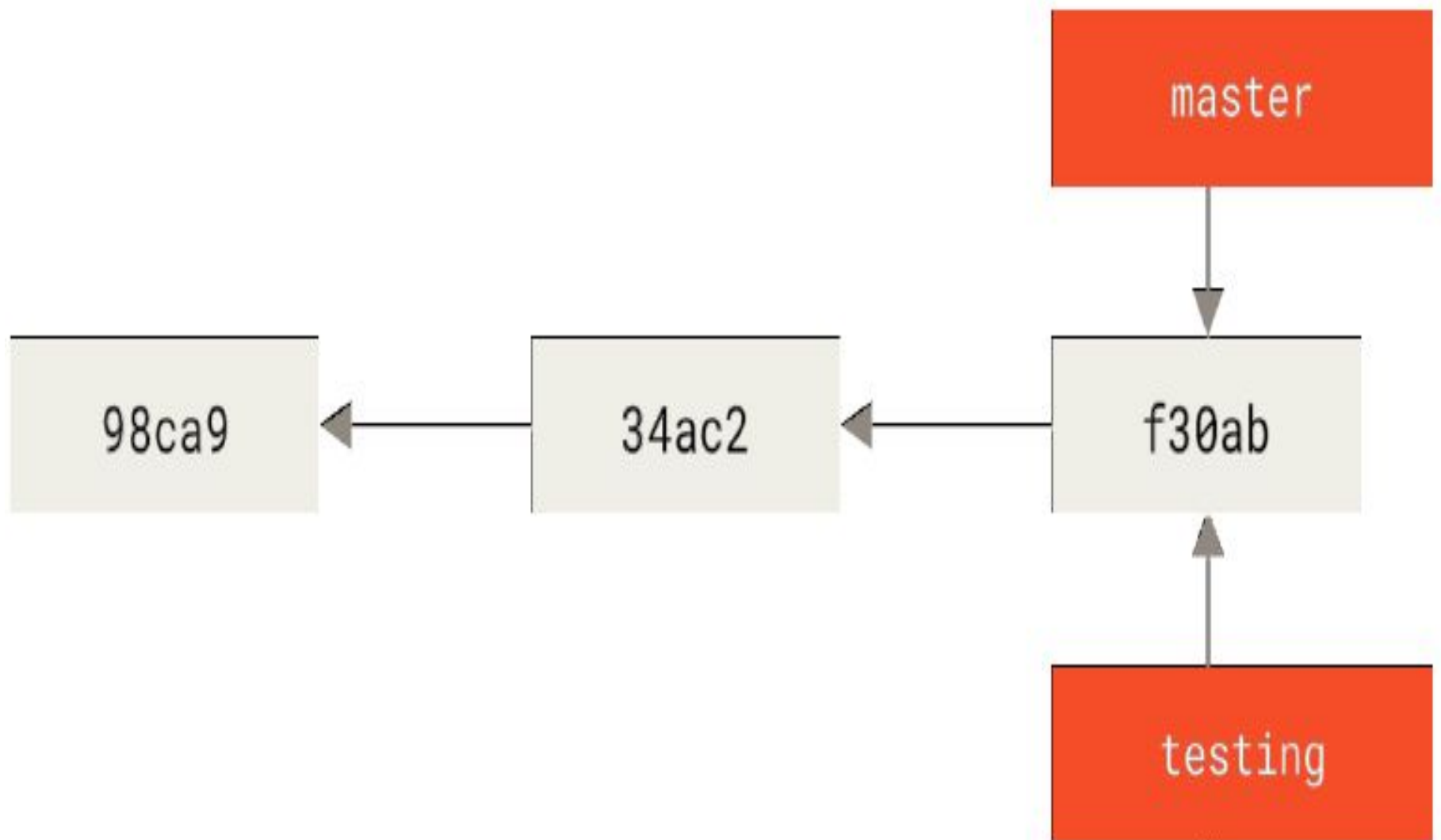


Figure 12. Two branches pointing into the same series of commits

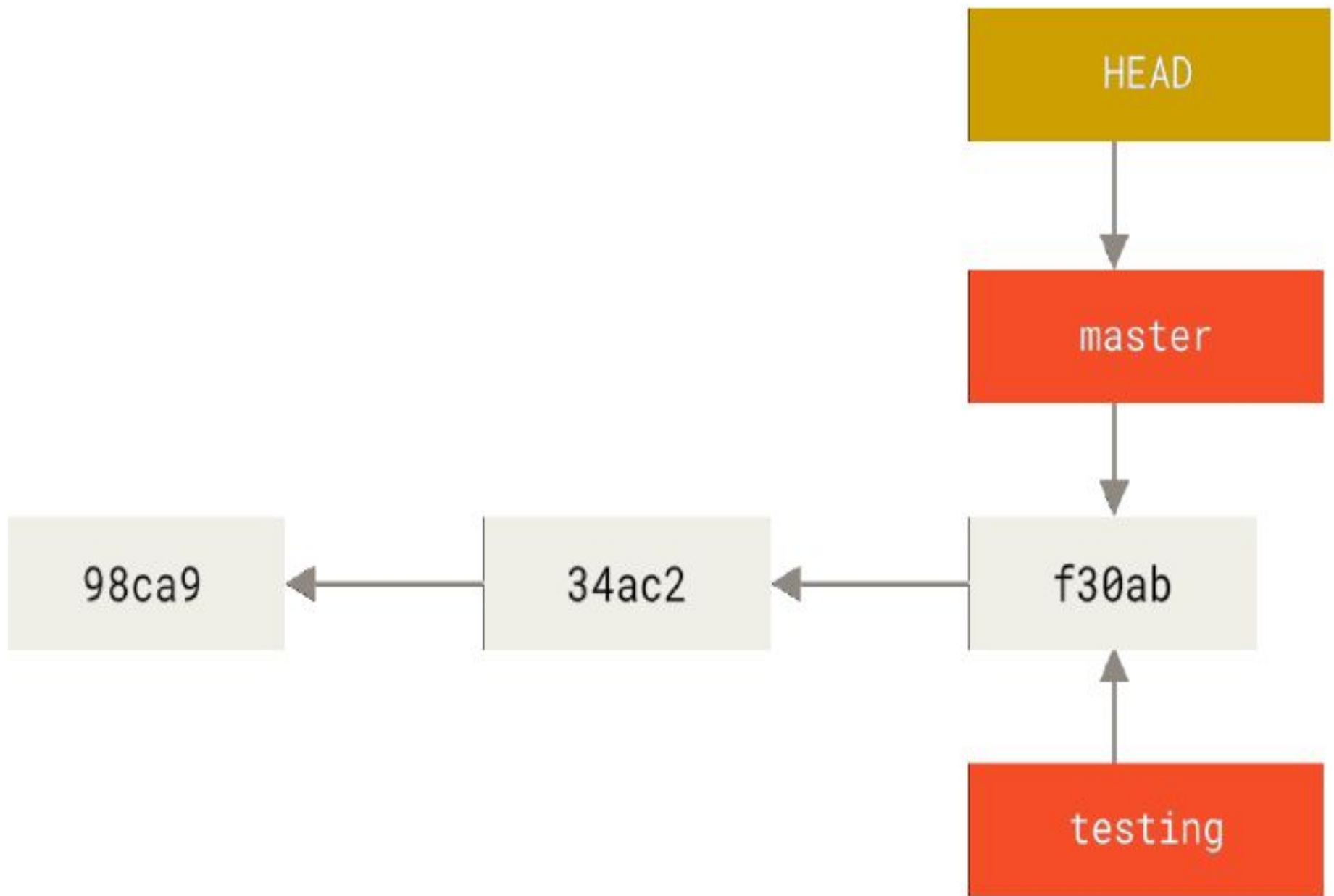


Figure 13. HEAD pointing to a branch

- It keeps a special pointer called HEAD.
- In Git, this is a pointer to the local branch you're currently on. Its still on master.
- The git branch command only *created a new branch,it didn't switch to that branch.*
- git log command that shows you where the branch pointers are pointing is called --decorate.

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

```
f30ab (HEAD -> master, testing) Add feature #32 - ability to add new formats to the  
central interface
```

```
34ac2 Fix bug #1328 - stack overflow under certain conditions
```

```
98ca9 Initial commit
```

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.

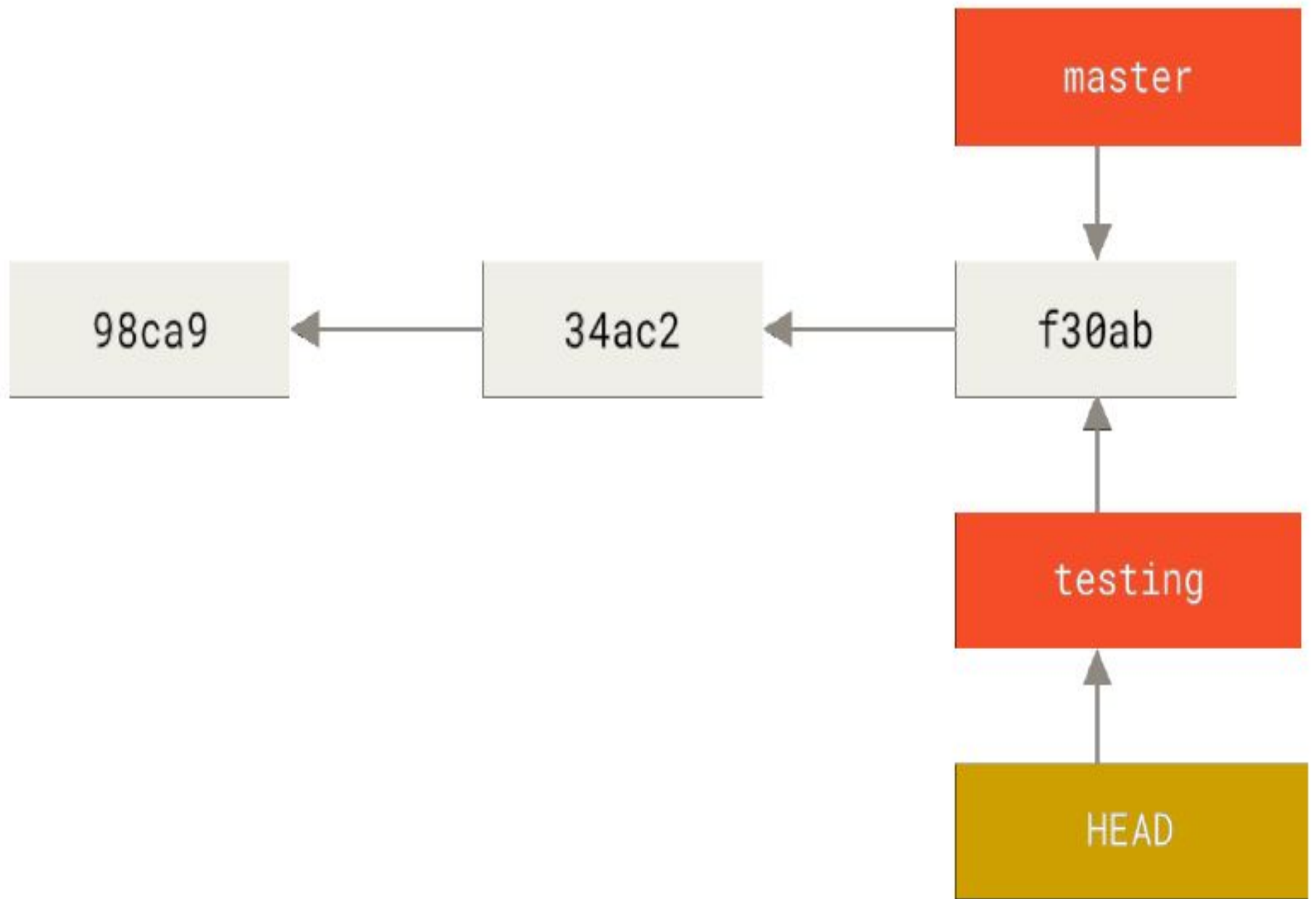


Figure 14. HEAD points to the current branch


```
$ vim test.rb
```

```
$ git commit -a -m 'made a change'
```

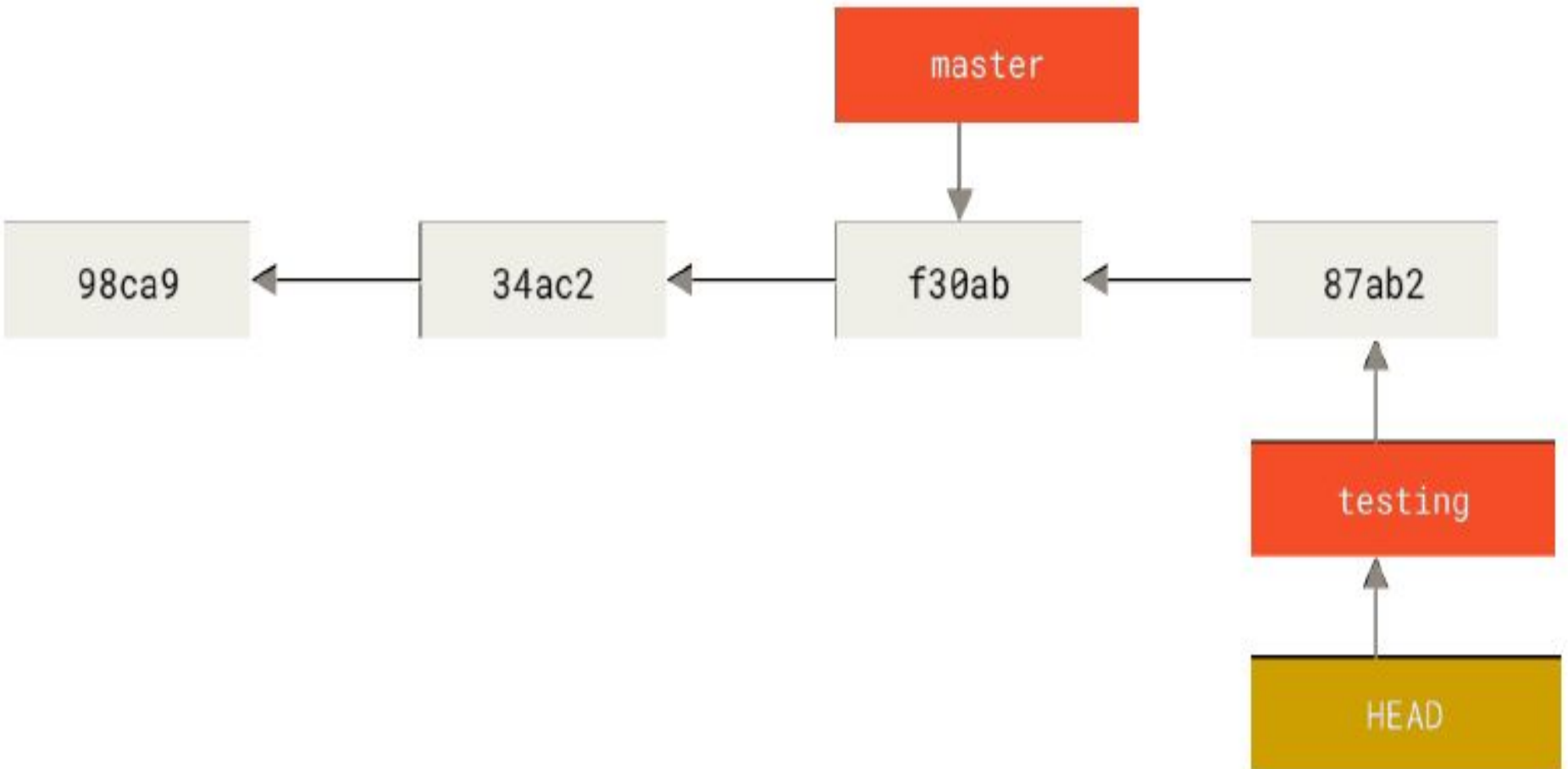


Figure 15. The HEAD branch moves forward when a commit is made

- testing branch has moved forward, but master branch still points to the commit you were on when you ran git checkout to switch branches. Let's switch back to the master branch:

\$ git checkout master

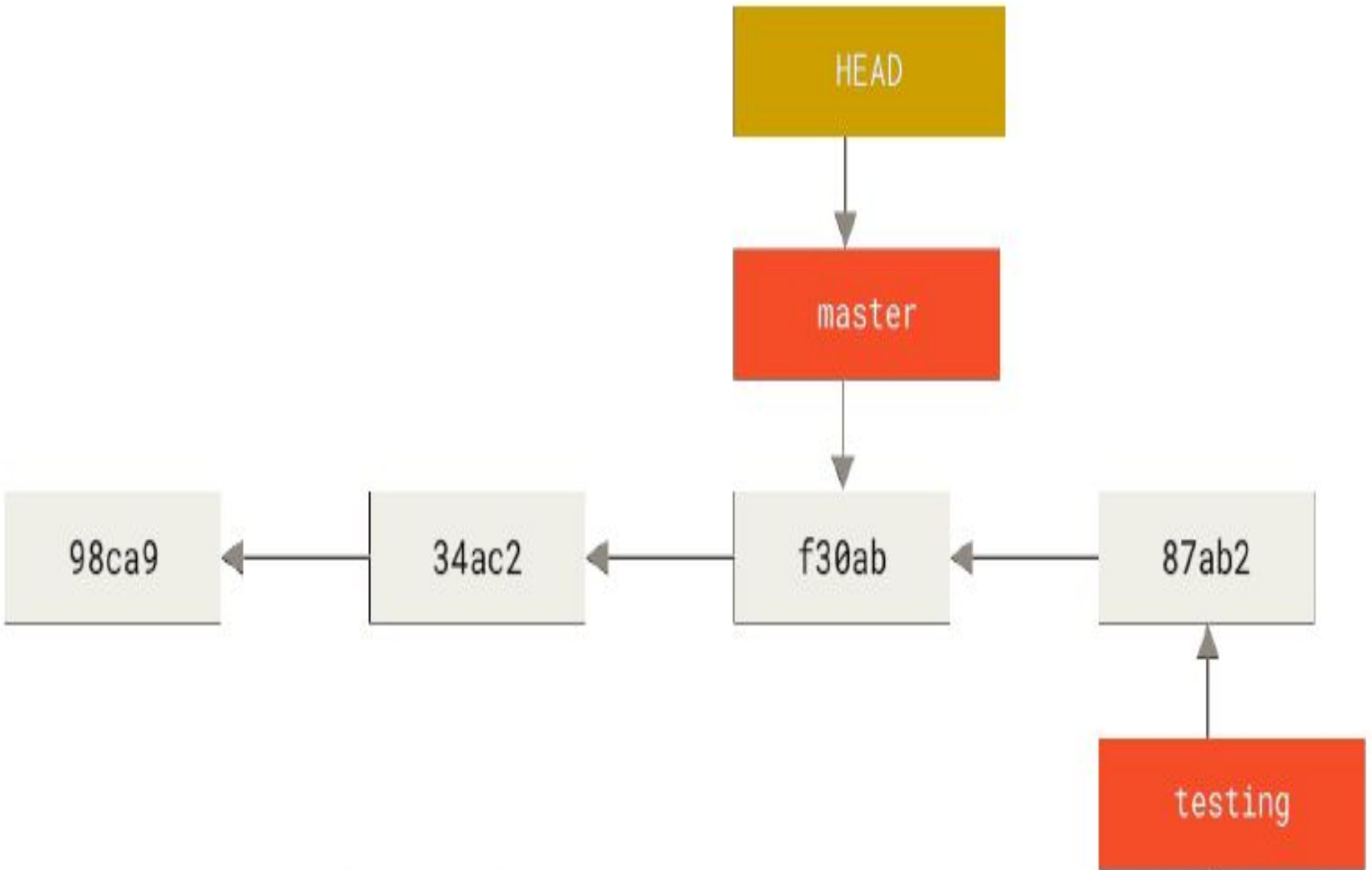


Figure 16. HEAD moves when you checkout

```
$ vim test.rb
```

```
$ git commit -a -m 'made other changes'
```

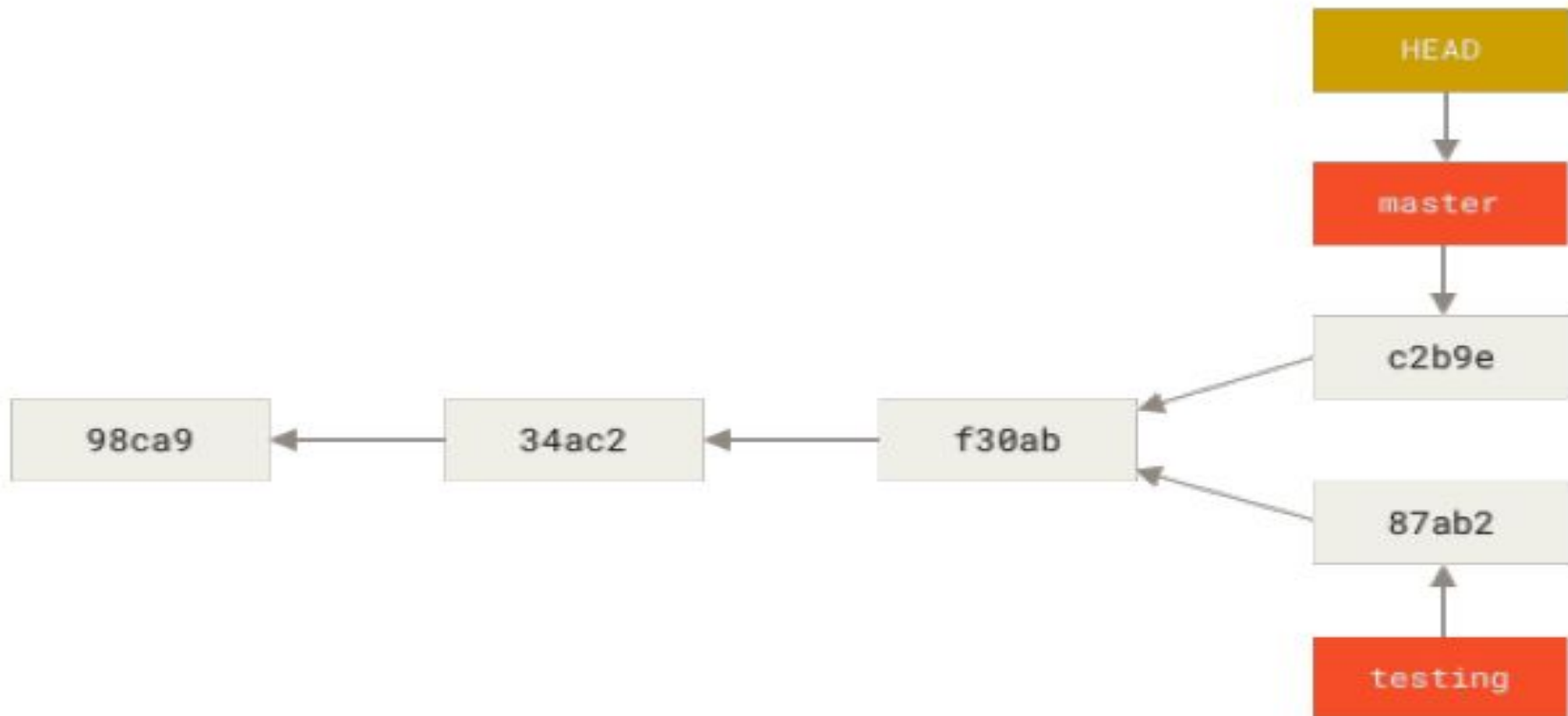


Figure 17. Divergent history

Push Changes

- When you do actions in Git, nearly all of them only *add data to the Git database*.
- As with any VCS, you can lose or mess up changes you haven't committed yet, but after you commit a snapshot into Git, it is very difficult to lose, especially if you regularly push your database to another repository.

```
# on John's computer
$ cd myproject
$ git init
$ git add .
$ git commit -m 'Initial commit'
$ git remote add origin git@gitserver:/srv/git/project.git
$ git push origin master
```

At this point, the others can clone it down and push changes back up just as easily:

```
$ git clone git@gitserver:/srv/git/project.git
$ cd project
$ vim README
$ git commit -am 'Fix for README file'
$ git push origin master
```


Adding Remote Repositories

- The `git add` command adds content from the working directory into the staging area (or “index”) for the next commit.
- When the `git commit` command is run, by default it only looks at this staging area, so `git add` is used to craft what exactly you would like your next commit snapshot to look like.

git add

- The ***git add*** command adds content from the working directory into the staging area (or “index”) for the next commit.
- When the ***git commit*** command is run, by default it only looks at this staging area, so ***git add*** is used to craft what exactly you would like your next commit snapshot to look like.

```
sngist@sngist-PC MINGW32 ~
```

```
$ cd test
```

```
sngist@sngist-PC MINGW32 ~/test (master)
```

```
$ git config --global user.name"reashmaraj"
```

```
sngist@sngist-PC MINGW32 ~/test (master)
```

```
$ git config --global user.email"reashmaraj.p.r@gmail.com"
```

```
sngist@sngist-PC MINGW32 ~/test (master)
```

```
$ git init
```

```
Reinitialized existing Git repository in C:/Users/sngist/test/.git/
```

```
sngist@sngist-PC MINGW32 ~/test (master)
```

```
$ git add final.xls
```

```
sngist@sngist-PC MINGW32 ~/test (master)
```

```
$ git commit -m "Commit done"
```

```
[master e38ad54] Commit done
```

```
1 file changed, 0 insertions(+), 0 deletions(-)
```

```
create mode 100644 final.xls
```

```
sngist@sngist-PC MINGW32 ~/test (master)
```

```
$ git remote add origin https://github.com/reashmaraj/myasetutorial
```

```
fatal: remote origin already exists.
```



```
sngist@sngist-PC MINGW32 ~/test (master)
```

```
$ gitgit commit -m "Commit done"
```

```
bash: gitgit: command not found
```

```
sngist@sngist-PC MINGW32 ~/test (master)
```

```
$ git commit -m "Commit done"
```

```
[master e38ad54] Commit done
```

```
1 file changed, 0 insertions(+), 0 deletions(-)
```

```
create mode 100644 final.xls
```

```
sngist@sngist-PC MINGW32 ~/test (master)
```

```
$ git remote add origin https://github.com/reashmaraj/myasetutorial
```

```
fatal: remote origin already exists.
```

```
sngist@sngist-PC MINGW32 ~/test (master)
```

```
$ git push -u origin master
```

```
Logon failed, use ctrl+c to cancel basic credential prompt.
```

```
Enumerating objects: 4, done.
```

```
Counting objects: 100% (4/4), done.
```

```
Delta compression using up to 4 threads
```

```
Compressing objects: 100% (3/3), done.
```

```
Writing objects: 100% (3/3), 11.70 KiB | 3.90 MiB/s, done.
```

```
Total 3 (delta 0), reused 0 (delta 0)
```

```
To https://github.com/reashmaraj/myasetutorial
```

```
017d9d4..e38ad54 master -> master
```

```
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

```
sngist@sngist-PC MINGW32 ~/test (master)
```

```
$ |
```



git diff

- The git diff command is used when you want to see differences between any two trees.
- This could be the difference between your working environment and your staging area, between your staging area and your last commit, or between two commits.

Conflict-resolution

- Git adds standard conflict-resolution markers to the files that have conflicts, so you can open them manually and resolve those conflicts.
- Your file contains a section that looks something like this:

```
<<<<<<< HEAD:index.html
```

```
<div id="footer">contact : email.support@github.com</div>
```

```
=====
```

```
<div id="footer">
```

```
  please contact us at support@github.com
```

```
</div>
```

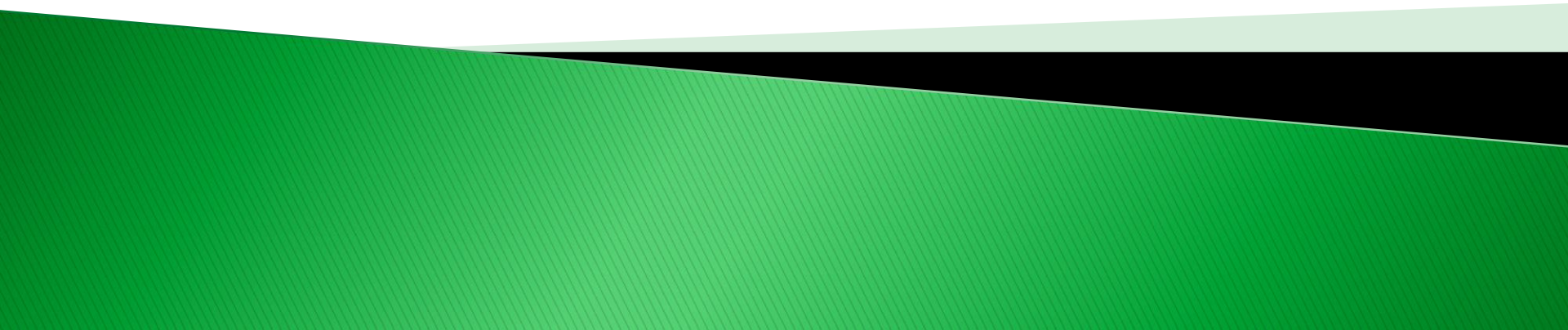
```
>>>>>>> iss53:index.html
```


- This means the version in HEAD is the top part of that block (everything above the =====), while the version in your iss53 branch looks like everything in the bottom part.
- In order to resolve the conflict, you have to either choose one side or the other or merge the contents yourself.
- For instance, you might resolve this conflict by replacing the entire block with this:

```
<div id="footer">  
please contact us at email.support@github.com  
</div>
```

Module 2 (Part III)

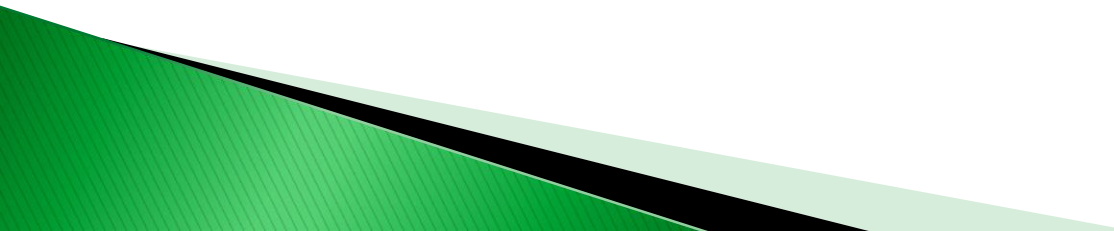
Software Quality



INDEX


- ▶ **WHAT IS QUALITY?**
- ▶ **FOUR DIMENSIONS OF QUALITY**
 - ❖ **Specification quality**
 - ❖ **Design quality**
 - ❖ **Development (software construction) quality**
 - ❖ **Conformance quality**

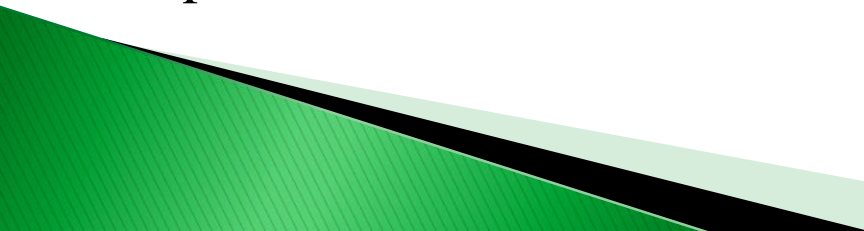
❖ WHAT IS QUALITY?


- ▶ To say that a certain product is a *quality product* implies that the product is of good quality
 - ▶ On the other hand, people certainly use the term *bad quality* to express their dissatisfaction with the products or services they use
 - ▶ Therefore, the adjective *good* is implicitly attached to the word *quality* in the minds of most people
 - ▶ Thus, the word *quality* connotes *good quality* to most people, including technical professionals
- 

▶ Before attempting a more elaborate definition of quality, let us consider the various connotations the word invokes, as it means different things in different sections of society:

- 1) For a **customer or end user** of a product, quality connotes defect-free functioning, reliability, ease of use
- 2) For a **producer of goods**, quality connotes conformance of the product to specifications
- 3) For a **provider of services**, quality connotes meeting deadlines and delivery of service that conforms to customer specifications and standards
- 4) For **government bodies**, quality connotes safety and protection of consumers from fraud
- 5) For an **industry association or standards body**, quality connotes safeguarding the industry's reputation, protecting the industry from fraud

- ▶ The International Organization for Standardization (ISO 9000, second edition, 2000) defines quality as the **degree** to which a set of inherent **characteristics** fulfills **requirements**
 - ▶ Quality can be used with such adjectives as poor, good, or excellent
 - ▶ This definition contains three key terms: **requirements, characteristics, and degree**
 - ▶ **Requirements** can be stated by a customer as product specifications
 - ▶ **Characteristics** refers to the capability of the deliverable
 - ▶ The word **degree** implies that quality is a continuum, beginning with zero and moving toward, perhaps, infinity
- 

- ▶ **Quality** is an attribute of a **product** or **service** provided to consumers that conforms the best of the **available specifications** for that product or service
 - ▶ It includes making **those specifications** available to the end user of the product or service
 - ▶ The specifications that form the basis of the product or service provided may have been defined by a government body, an industry association, or a standards body
 - ▶ Where such a definition is not available, the provider may define the specifications
- 

- ▶ The result of a product or service that meets the above definition of quality is that the customer is able **to effectively use the product** for the length of its life or enjoy the service fully
 - ▶ This result further mandates that the **provider is responsible for providing any support** that is required by the customer
 - ▶ Any product or service that meets the requirements of this definition is rated a “**quality product/service**”
 - ▶ Any product or service that does not meet the requirements of this definition is rated “**poor quality**”
 - ▶ **Reliability** of a product is its **capability to function at the defined level** of performance for the duration of its life
- 

❖ **FOUR DIMENSIONS OF QUALITY**


- ▶ Quality has four dimensions

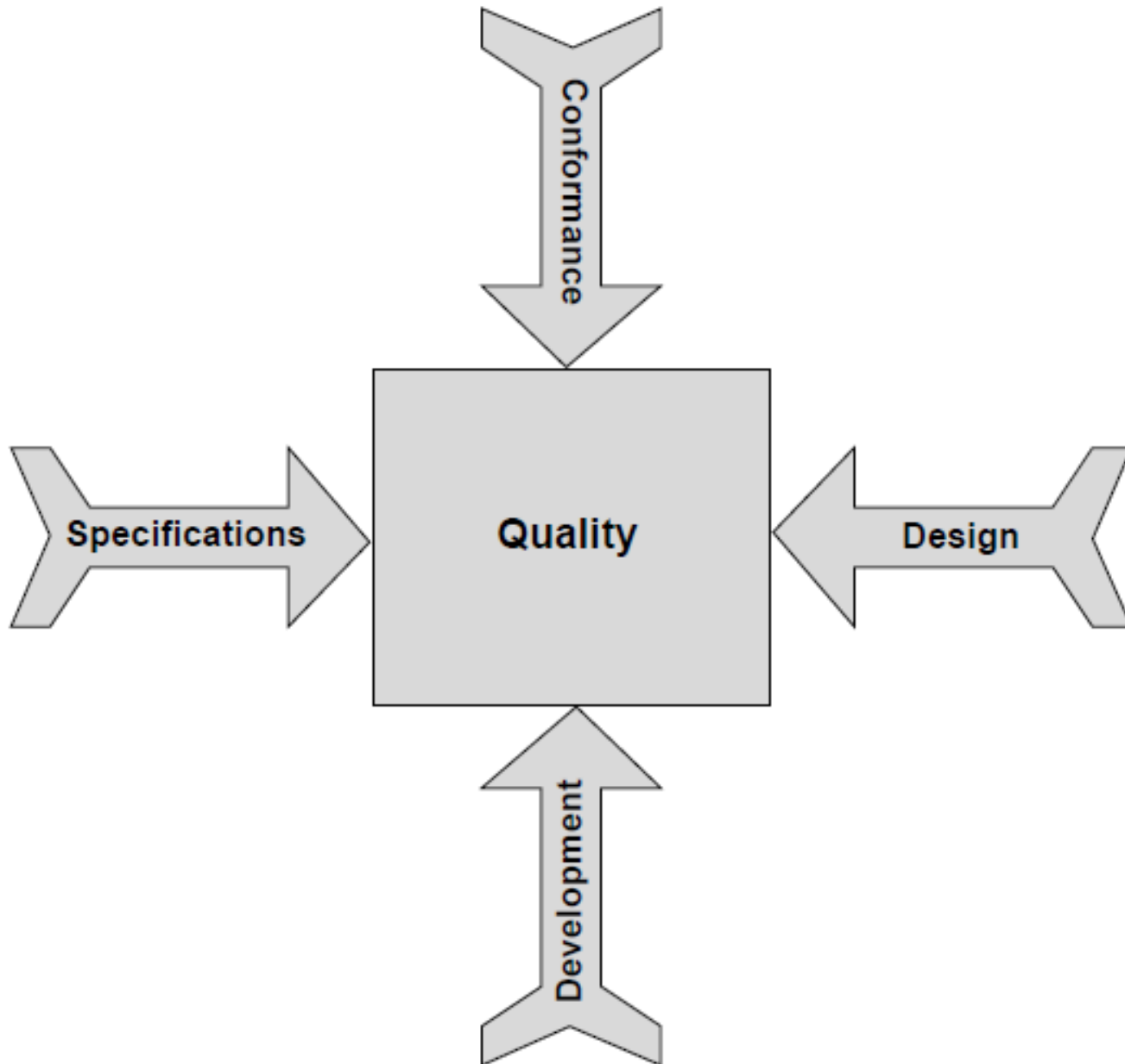
- ❖ **Specification quality**

- ❖ **Design quality**

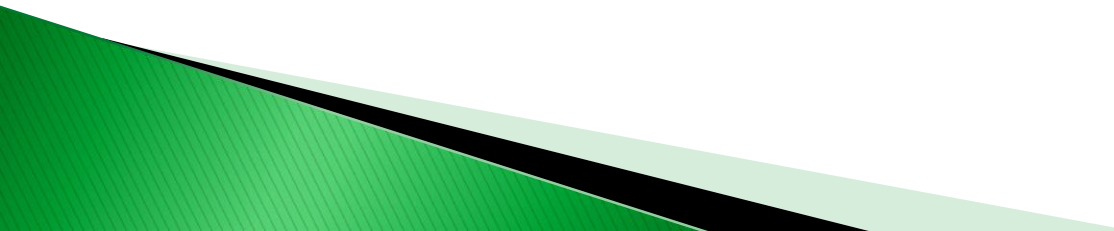
- ❖ **Development (software construction) quality**


- ❖ **Conformance quality**

- ▶ Specifications are the starting point in the journey of providing a product or service, followed by design and then development
 - ▶ Conformance quality is ensuring how well that quality is built into the deliverable at every stage
- 

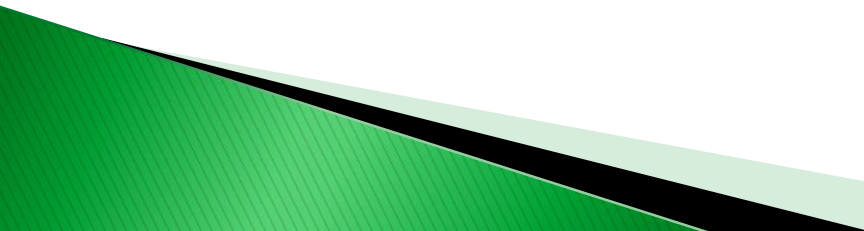


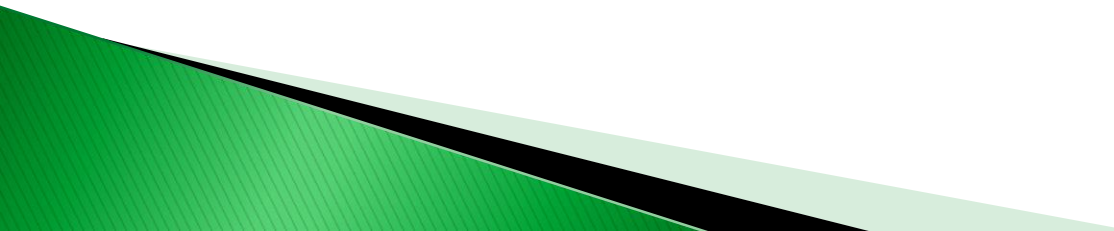
1) Specification Quality

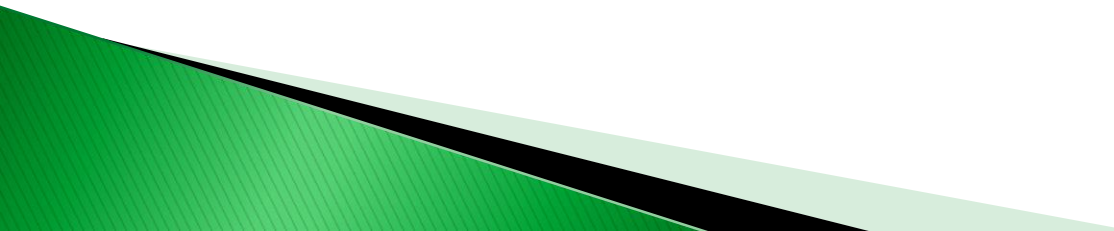
- ▶ Specification quality refers to **how well the specifications are defined** for the product or service being provided
 - ▶ Specifications have no predecessor activity, and all other activities succeed specifications
 - ▶ Thus, **if the specifications are weak, design will be weak**, resulting in the development and manufacture of an incorrect product, and the effort spent on ensuring that quality is built in will have been wasted
- 

- ▶ Specifications normally should include the following six aspects:
 - 1) **Functionality aspects** : Specify **what functions** are to be achieved by the product or service
 - 2) **Capacity aspects** : Specify the **load the product can carry** (such as 250 passengers on a plane or 100 concurrent users for a Web application)
 - 3) **Intended use aspects** : Specify the **need or needs the product or service** satisfies
 - 4) **Reliability aspects** : Specify **how long the product can be enjoyed** before it needs maintenance
 - 5) **Safety aspects** : Specify **the threshold levels for ensuring safety to persons** and property from use of the product or service
 - 6) **Security aspects** : Specify any **threats** for which the product or service needs to be prepared
- 

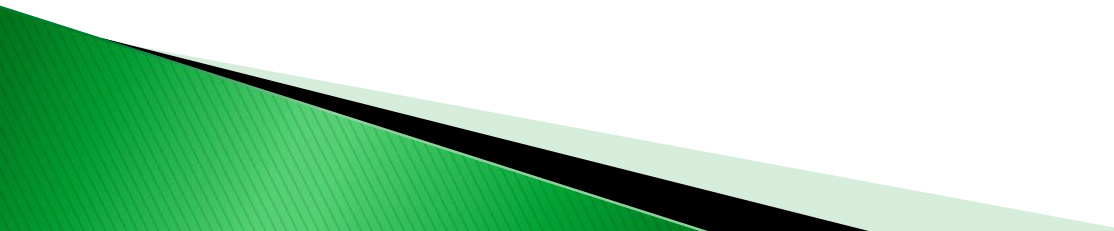
➤ Ensuring Quality in Specifications

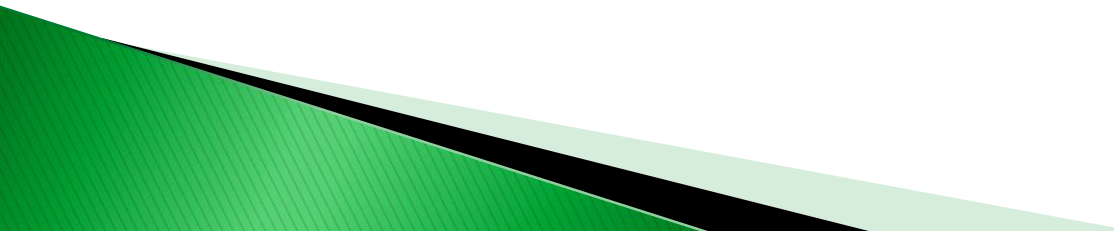
- ▶ In the software industry, specifications are referred to as **user requirements**
 - ▶ The following are possible scenarios for obtaining user requirements:
 - 1) A business analyst conducts a **feasibility study, writes up a report, and draws up the user requirements**. The analyst:
 - a) Meets with all the **end users** and notes their requirements and concerns
 - b) Meets with the **function heads** and notes their requirements and concerns
 - c) Meets with **management personnel** and notes their requirements and concerns
 - d) **Consolidates the requirements** and presents them to select end users, function heads, and management personnel and receives their feedback, if any
 - e) **Implements the feedback** and finalizes specifications
- 


- 2) A **ready set of user requirements** is presented as part of a request for proposal
 - 3) A **request for proposal** points to a **similar product** and requests replication with client-specific customization
- ▶ Regardless of the scenario, once the specifications are ready, **quality assurance** steps in
 - ▶ The role of quality assurance in this area is to ensure that the specifications are exhaustive and cover all areas
 - ▶ Including **functionality, capacity, reliability, safety, security, intended use**, etc.
- 

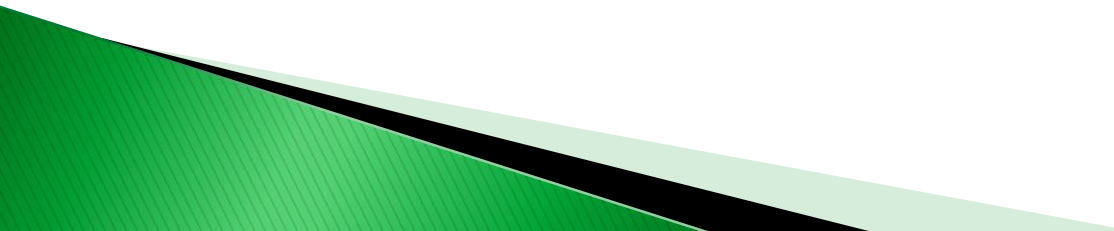
- ▶ The tools for building quality into specifications are as follows:
 - ❖ **Process documentation** - Details the methodology for gathering, developing, analyzing, and finalizing the specifications
 - ❖ **Standards and guidelines, formats, and templates** - Specify the minimum set of specifications that needs to be built in
 - ❖ **Checklists** - Help analysts to ensure comprehensiveness of the specifications
- 

2) Design Quality

- ▶ Design quality refers to **how well the product or service to be delivered is designed**
 - ▶ The objectives for design are to **fulfill the specifications** defined for the product or service being provided
 - ▶ Design determines the shape and strengths of the product or service
 - ▶ Therefore, **if the design is weak, the product or service will fail**, even if the specifications are very well defined
- 

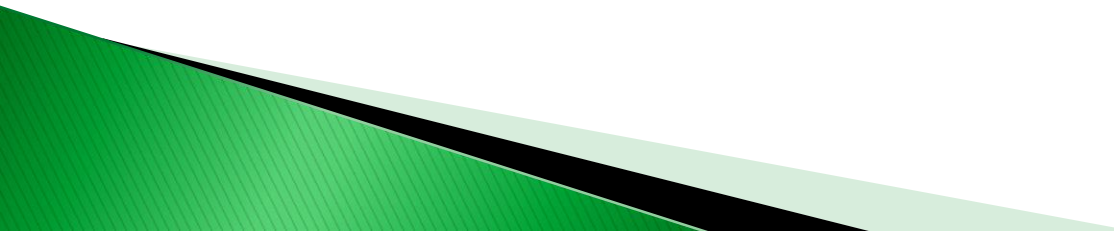
- ▶ Design can be split into two phases: **conceptual design and engineering**
 - ▶ *Conceptual design* selects the approach to a solution from the multiple approaches available
 - ▶ *Engineering* uses the approach selected and works out the details to realize the solution
 - ▶ Conceptual design is the creative part of the process, and engineering is the details part
 - ▶ In terms of software, conceptual design refers to **software architecture**, navigation, number of tiers, approaches to flexibility, portability, maintainability, and so on
 - ▶ Engineering design refers to **database design**, program specifications, screen design, report design, etc.
- 

- ▶ Software design normally contains the following elements:
 - 1) Functionality design
 - 2) Software architecture
 - 3) Navigation
 - 4) Database design
 - 5) Development platform
 - 6) Deployment platform
 - 7) User interface design
 - 8) Report design
 - 9) Security
 - 10) Fault tolerance
 - 11) Capacity
 - 12) Reliability
 - 13) Maintainability
 - 14) Efficiency and concurrence
 - 15) Coupling and cohesion
 - 16) Program specifications
 - 17) Test design
- 

- ▶ It is normal to conduct a *brainstorming session* at the beginning of a software design project, to select **one optimum design alternative** and to decide on the overall design aspects
 - ▶ Such as the number of tiers, technology platform, software coupling and cohesion, etc.
 - ▶ A brainstorming session helps designers arrive at the **best possible solution** for the project at hand
- 

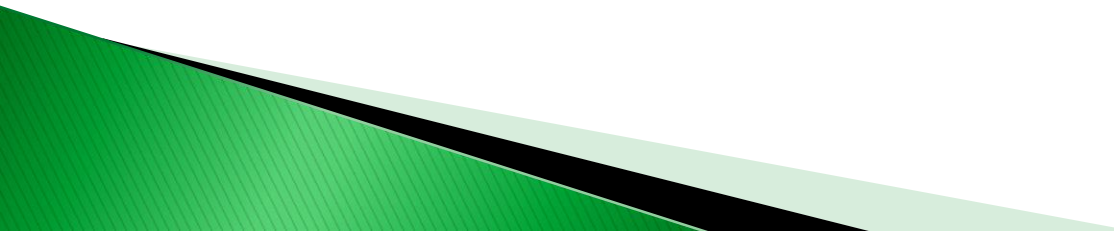
➤ Ensuring Quality in Design


- ▶ Normally, software design is a two-step process:
 - **Conceptual design** - Referred to as **high-level design**, functional design specification, software requirements specification, and software architecture design
 - **Engineering design** - Referred to as **low-level design**, detailed design specification, software design description, and software program design

- ▶ The tools for building quality into design include the following:
 - ❖ **Process documentation** - Details the methodology for **design alternatives** to be considered, **criteria** for selecting the alternative for the project, and finalizing the **conceptual design**
 - ❖ **Standards and guidelines, formats, and templates** - Specify the **possible software architectures** along with their attendant advantages and disadvantages and so on
 - ❖ **Checklists** - Help designers to ensure that **design** is carried out comprehensively and appropriately
- 

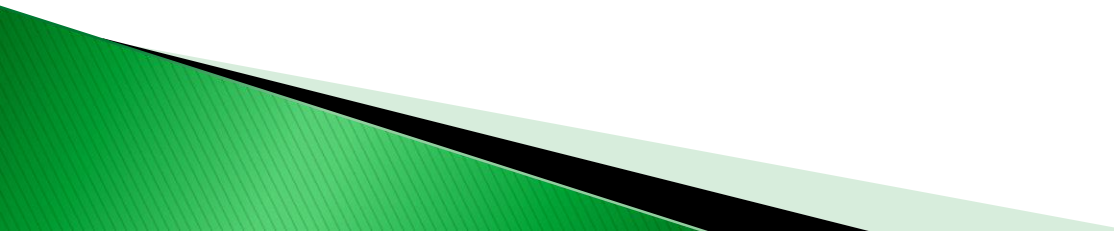
3) Development (software construction) Quality

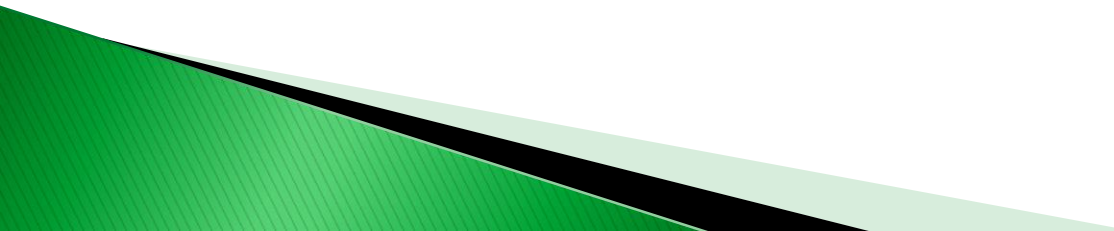
The following activities form part of developing software:

- Create the database and table structures
 - Develop dynamically linked libraries for common routines
 - Develop screens
 - Develop reports
 - Develop unit test plans
 - Develop associated process routines for all other aspects, such as security, efficiency, fault tolerance, etc.
- 

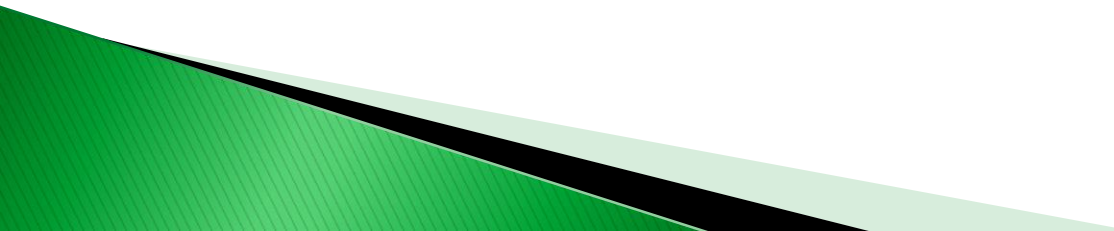
- ▶ Good-quality construction is achieved by adhering to the ***coding guidelines*** of the programming language being used
 - ▶ Normally there is a separate coding guideline for every programming language used in an organization
 - ▶ Coding guidelines **contain naming conventions, code formatting** that help developers write reliable and defect-free code
 - ▶ Of course, it is very important to have **qualified people** trained in software development
 - ▶ Construction follows software design, and it should **always conform to the design document**
 - ▶ In this way, good quality in construction can be achieved
- 

➤ Ensuring Quality in Development (Software Construction)

- ▶ Quality is built in by adhering to the **organizational standards** for code quality as well as the coding guidelines for the development language being used
 - ▶ Uncontrolled changes can wreak havoc with code quality
 - ▶ Therefore, change management and configuration management assume importance for ensuring code quality
- 

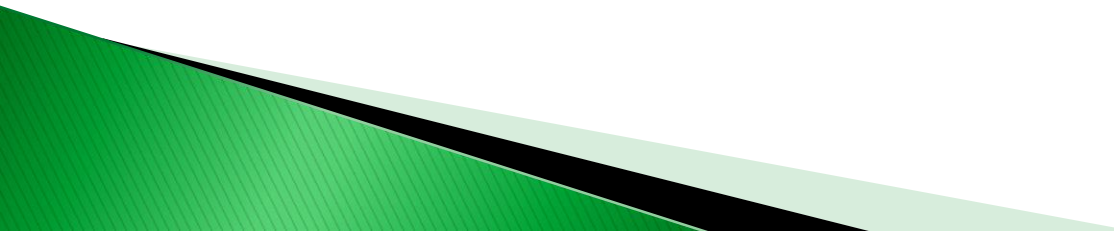
- ▶ There are two techniques to ensure that quality is built into a product:
 - Reviews (walkthroughs)
 - Testing
- 

4) Conformance Quality

- ▶ Conformance quality deals with **how well an organization ensures that quality is built into a product** through the above three dimensions
 - ▶ It is one thing to do a **quality job**
 - ▶ But it is quite another to **unearth any defects lurking in the work product** and ensure that **a good-quality product is indeed built**
 - ▶ Essentially, conformance quality examines how well **quality control** is carried out in the organization
- 

➤ Ensuring Conformance Quality

- ▶ Ensuring that conformance quality is at desirable levels in the organization is achieved through :
 - Audits
 - Quality measurements
 - Metrics
 - Benchmarking
- ▶ Defect removal efficiency of verification and validation activities, defect injection rate, and defect density are all used for this purpose

- ▶ **Audits** also are conducted to ensure that projects conform to various applicable standards for building quality into all activities, including specifications and design
 - ▶ In addition, organizational data is **benchmarked** against industry benchmarks, and corrective or preventive actions are taken to ensure that organizational conformance is indeed on a par with the industry
 - ▶ Conformance quality is built in through **process definition** and **continuous improvement** for all software development activities as well as **quality assurance**
- 

Quality dimension	How to build in quality	Techniques for ensuring quality
Quality of specifications	Specification development process documentation; standards and guidelines, formats, and templates for defining specifications; and checklists	Expert reviews, peer reviews, and brainstorming
Quality of design	Software design process documentation; standards and guidelines, formats, and templates for software design; and checklists	Expert reviews, peer reviews, managerial reviews, and brainstorming
Quality of development	Coding guidelines, configuration management, and change management	Peer reviews and software testing
Conformance quality	Diligent application of all quality assurance activities in the organization, process definition, and improvement	Audits, measurement and metrics for quality assurance activities, and benchmarking of organizational metrics against industry metrics