Dept. of Meteorology, School of Mathematical and Physical Sciences

University of Reading

Scenario DOCTORAL TRAINING PARTNERSHIP

Software Development for Environmental Scientists

Level 2 : 21st May, 4th & 11th June 2018

18 May 2018

www.reading.ac.uk

University of Reading

Introductions

- Housekeeping, refreshments/lunch, facilities
- Your course instructor: Jane
- You
  - Name;
  - How much do you know about…
    - Distributed version control
    - Object oriented design
    - Automated testing

University of Reading

Course Overview (1)

- Day 1
  - Version Control
    - Revision
    - Team working
    - GitHub
  - OO introduction
    - OO concepts
- Day 2
  - Revision
  - OO Design & CRC
  - Diagrams
  - More OO concepts
  - Standards & Naming

University of Reading

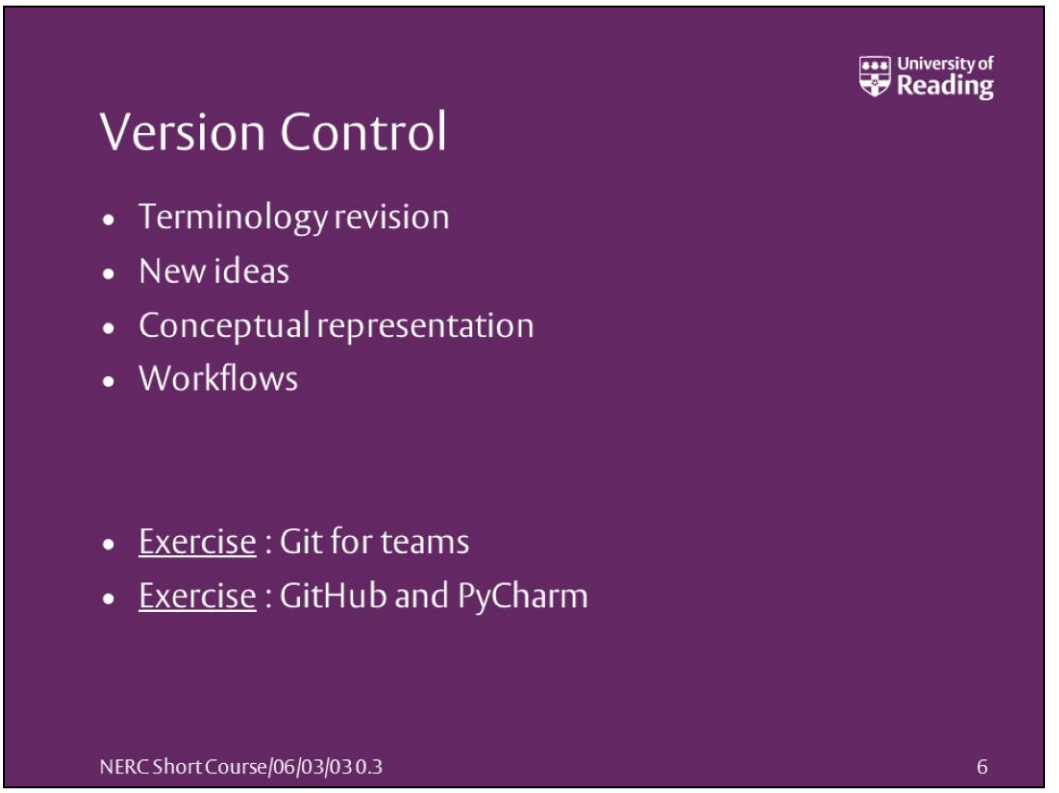
Course Overview (2)

- Day 3
  - Exception classes
  - Testing
  - Design patterns
  - Debugging
  - Admin
  - Q & A
- Annex
  - XML
  - JSON
  - Software Process
  - MPI

University of Reading

Software Development for Environmental Scientists

DAY 1



Terminology revision
init, status, log, add, commit, branch, checkout

New terms
clone, remote, fetch, rebase, merge, push

Workflows
https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow
http://git-scm.com/book/en/v2/Distributed-Git-Distributed-Workflows
https://help.github.com/articles/what-is-a-good-git-workflow/
http://blog.endpoint.com/2014/05/git-workflows-that-work.html

Revision of terms

Repository

The (backed-up) central store of code which contains all the information to get any version, keep track of changes and do all the housekeeping. It can be locally on your machine, on a server, or on a cloud somewhere (e.g. GitHub).

Local repository

Some version control systems work by making a copy of the central repository on your machine where you do local version control but then there are extra steps to ensure it's synchronised with the central repo. This is a really safe configuration though brings extra levels of complexity.

Main / trunk

This is the original code, the set of files upon which changes are made in the normal course of events. You can add more files and delete old ones. Note that deleting a file only deletes it from that version of the repository, it'll still be there if you go back to an earlier version (unless you're very sneaky).

Branch

You can take a branch from the trunk to make experimental changes, or to do something new independent of what's going on in the main codebase – effectively you've speciated!

Checkout

Some systems require a definitive action to lock the file, or set of files, against change by someone else. This can be limiting because you're stopping anyone else even fixing a bug.

Check-in

You want to commit your changes to the repository, once checked in you have a reference version and you can always go back to it. You can see what it was you changed by viewing the differences between files, or you can see the summary of alterations by looking at the sequence of check-in comments you've written. In a two tier system, you have to check in and then push the changes, the changes are no longer private and they are then shared with everyone else. Never check in something that breaks the program, always provide a check-in comment which says what you've done. If you need to check in to keep your changes safe but it doesn't work, consider disabling your new functionality.

Merge / Conflict resolution

Using a 2-tier model or having a central server, if you check in a changed file, either one that was from the trunk or a set from a branch, the version control system will tell you that there are more differences between the files than the changes that you alone have made – this happens when someone else has worked on the file in parallel and got there first with the check-in! Usually you'll be prompted to accept or reject each individual change from the two sources so that they are combined into a new version containing the merged changes. This can be really tricky and you are best to make a local copy of the merged file and test it before committing it.

Diff

Most version control tools allow you to see the differences between any two versions of a file, one of those versions may be your locally modified copy.

History

The version control system records every change to a file from one version to the next and will provide you with a diff between any pair of versions. It also records the check-in comment so that you can see what was going on or why the change(s) was made. The total changes to a file are its history.

Tag / label / baseline

Every so often, you want to be able to put a stake in the ground and say 'this works'! This is where you apply a marker to every single file in the repository so that you can recall the whole set, all the individual various versions, at any time. It's a really good idea to do this reasonably regularly, but do use sensible names. It's a very good idea to decide on a method for naming so it's clear what's a main release vs what's a working version prior to test and debugging.

Push

A variation on check-in; a version control system keeping tabs on your local working copy may allow you to check in locally, but then in addition you may have to push the changes to the central repository (see above).

Pull & update

Get all the latest code from the repository and copy it to your local working area *and* override any local changes you may have made.

Concepts

Subversion (http://tortoisesvn.net/docs/nightly/TortoiseSVN_en/tsvn-qs-basics.html)

"The Repository" Subversion uses a central database which contains all your version-controlled files with their complete history. This database is referred to as the *repository*. The repository normally lives on a file server running the Subversion server program, which supplies content to Subversion clients (like TortoiseSVN) on request. If you only back up one thing, back up your repository as it is the definitive master copy of all your data.

"Working Copy" This is where you do the real work. Every developer has his own working copy, sometimes known as a sandbox, on his local PC. You can pull down the latest version from the repository, work on it locally without affecting anyone else, then when you are happy with the changes you made commit them back to the repository.

"A Subversion working copy does not contain the history of the project, but it does keep a copy of the files as they exist in the repository before you started making changes. This means that it is easy to check exactly what changes you have made."

http://hginit.com/00.html : useful diagram

If you are using Windows, you might want to use TortoiseSVN which gives you

access to all the commands via a pretty GUI.

Update – get the latest from the central repo.

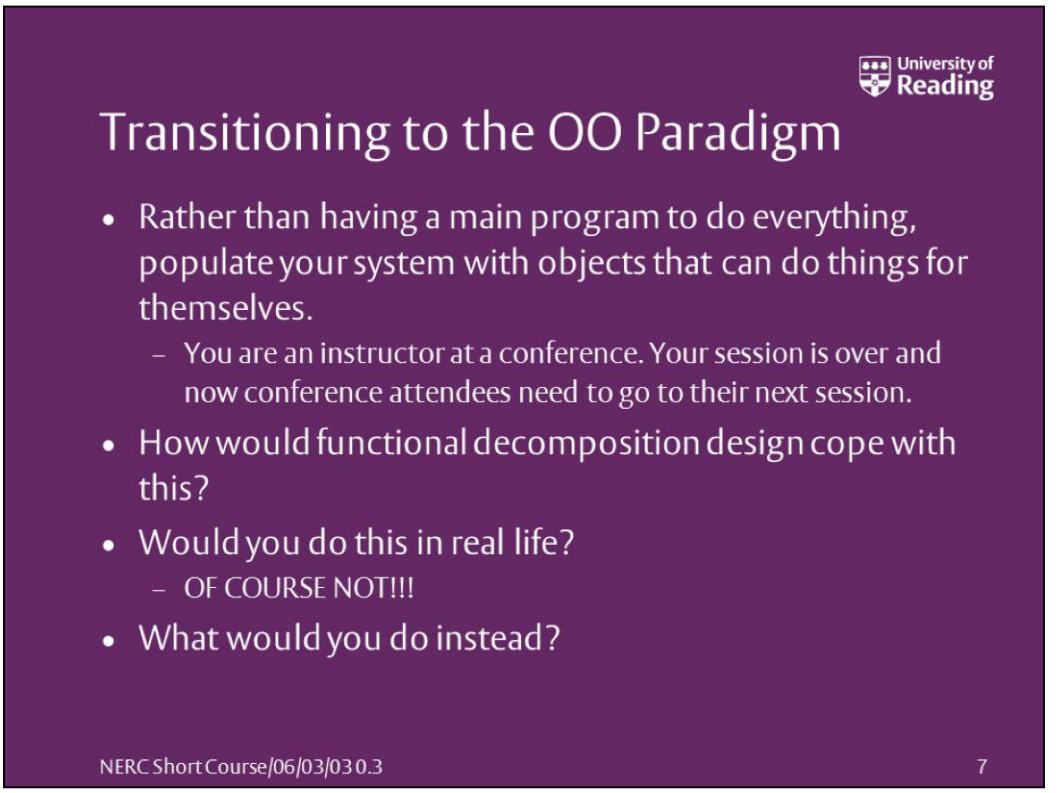
Commit – send your changes to the central repo., always use a sensible comment.

Also diff, branch, tag, merge are available of course.

Git (http://www.sbf5.com/~cduan/technical/git/)

Again, a repository is created which takes care of the versioning information for each file. However, there are two stages to checking in a file, this is because of the internal workings to allow for remote repositories and code sharing. All copies of the repository are of equal status though which is a different model compared with SVN. Often for team working, one copy of the repository is treated as the master and it's usually an online one (GitHub or BitBucket etc.)

For your private use, install git, initialise it, then add files (to get them ready), then commit them. Git has branches and diff too.



Scenario: You are an instructor at a conference. Your session is over and now conference attendees need to go to their next session

With functional decomposition, you would develop a program to solve this problem that would have you the instructor do everything:

get the roster, loop through each attendee, look up their next session,
find its location, generate a route, and, finally,
tell the attendee how to get to their next class

You would do everything, attendees would do (almost) nothing

What would you do instead?

You would assume that everyone has a conference program, knows where they need to be next, and will get their on their own. All you would do is end the session and head off to your next activity

At worst, you would have a list of the next sessions at the front of the class and you would tell everyone "use this info to locate your next session"

Compare / Contrast

In the first scenario, you know everything, you are responsible for everything, if something changes you would be responsible for handling it you give very explicit instructions to each entity in the system.

In the second scenario, you expect the other entities to be self sufficient you give very general instructions and you expect the other entities to know how to apply those general instructions to their specific situation.

Benefits of the second scenario

The biggest benefit is that entities of the system have their own responsibilities. indeed this approach represents a shift of responsibility away from a central control program to the entities themselves.

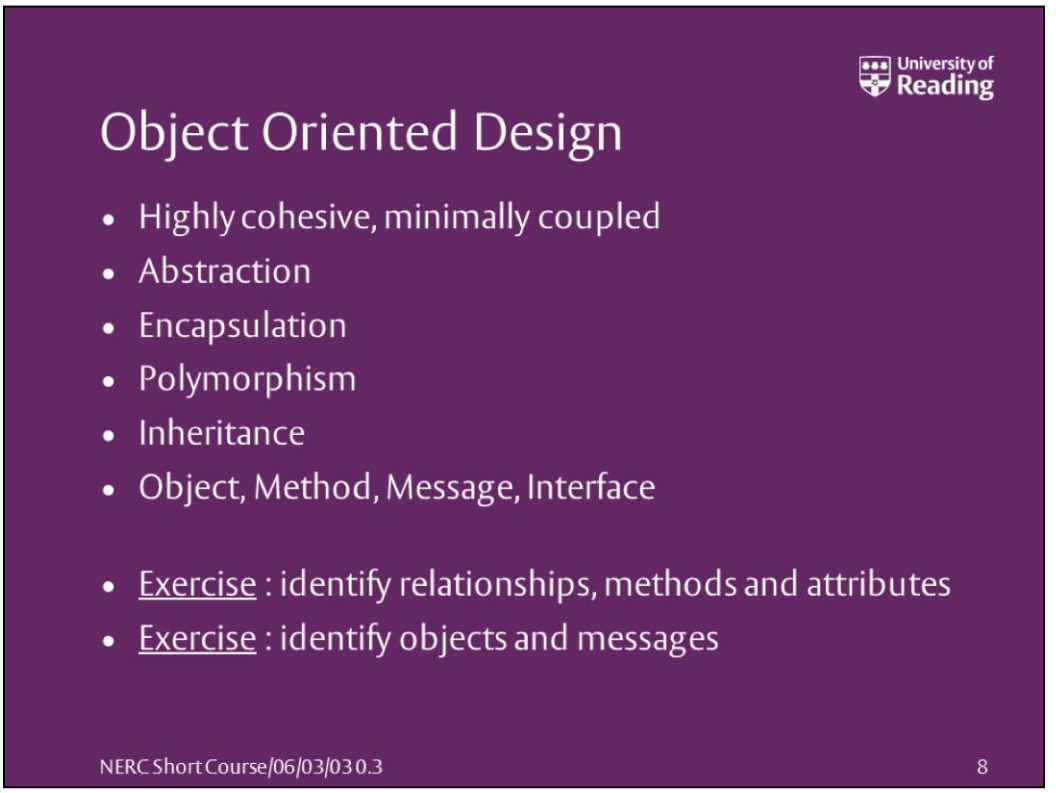
Suppose we had attendees and student volunteers in our session and that volunteers needed to do something special in between sessions:

> First approach: the session leader needs to know about the special case and remember to tell volunteers to do it before going to the next session.
>
> Second approach: the session leader tells each person "Go to your next session"; volunteers will then automatically handle the special case without the session leader needing to know anything about it.

We can add new types of attendees without impacting the leader.

(from http://www.cs.colorado.edu/~kena/classes/5448/f12/lectures/02-ooparadigm.pdf

which is highly recommended, no really, look at it!)



We need a shift in thinking to go from the functional decomposition design method to the object oriented design paradigm. Here we must think in terms of discrete entities which perform specific tasks, look after their own data and have a public face to interact with each other. Objects send messages to each other without any need to know how the message is processed or what internal data may be involved.

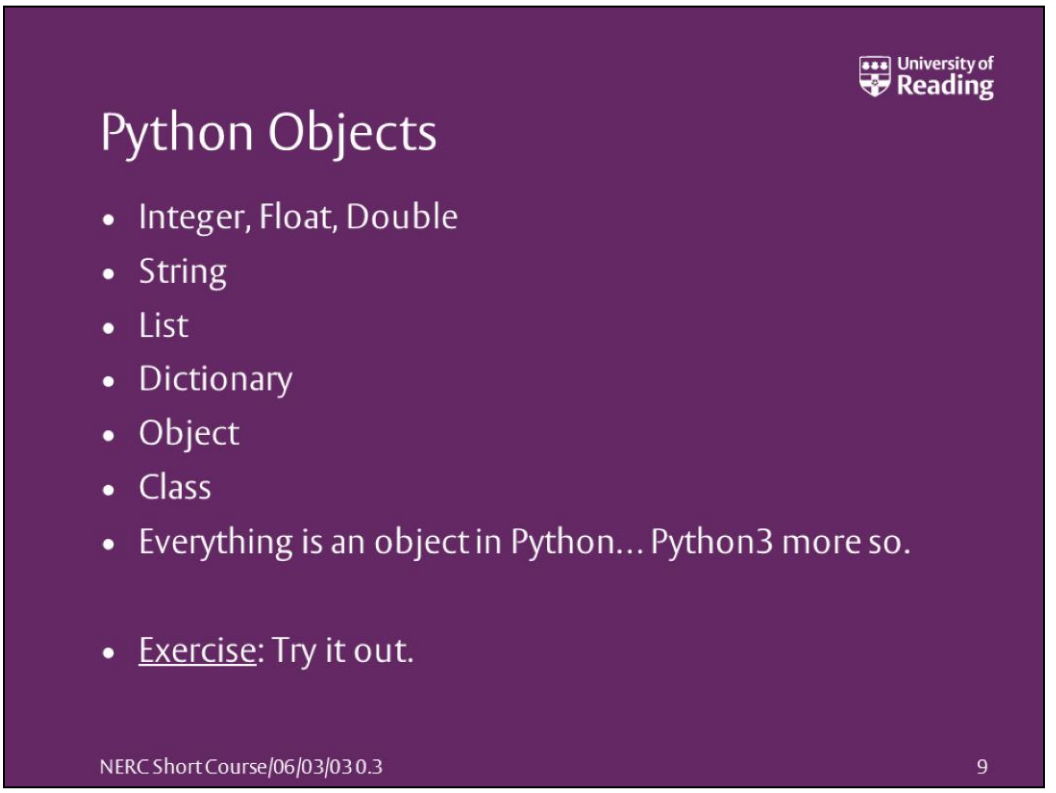
The data and methods are bound together, but the objects are more independent of each other resulting in a highly cohesive but loosely coupled design. It makes plugging in extra functionality much simpler.

The four paradigms of OO are:

- **Abstraction** – a set of concepts that some entity provides you in order to achieve a task
- **Encapsulation** - users see only the services available from an object, but not how those services are implemented
- **Polymorphism** - two or more objects respond to the same message but may do so in different ways
- **Inheritance** - allows an object to have the same behaviour as another and extend or tailor that behaviour to provide special action for specific needs

The basic definitions are:

- **Object** a collection of private data and a set of operations that can be performed on that data.
- **Class** the blueprint for an object such that many objects can be made with identical operations but varying data.
- **Method** an operation for accessing and manipulating data within an object.
- **Message** a request for a specific action to be performed by a specific object.
- **Interface** a collection of methods which specify how messages can be sent to an object.

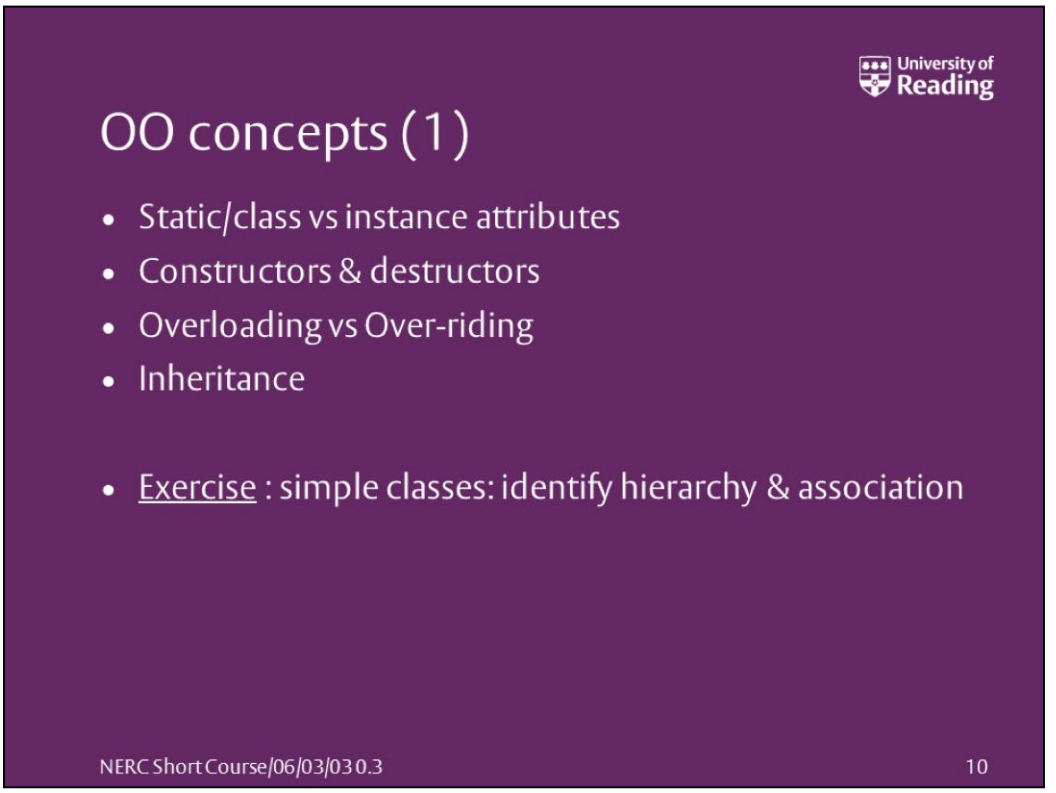


Some great Python resources:

http://www.diveintopython.net/

Thinking in terms of objects, what sort of things would you want to be able to do if you had a simple type?

And for a more complex type?



OO Terminology

Class: A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.

Class variable: A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables aren't used as frequently as instance variables are.

Data member: A class variable or instance variable that holds data associated with a class and its objects.

Function overloading: The assignment of more than one behaviour to a particular function. The operation performed varies by the types of objects (arguments) involved.

Instance variable: A variable that is defined inside a method and belongs only to the current instance of a class.

Inheritance : The transfer of the characteristics of a class to other classes that are derived from it.

Instance: An individual object of a certain class. An object that belongs to a class Circle, for example, is an instance of the class Circle.

Instantiation : The creation of an instance of a class.

Method : A special kind of function that is defined in a class definition.

Object : A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods.

Operator overloading: The assignment of more than one function to a particular operator.

This is really good and well worth reading thoroughly to get to grips with the syntax:

http://www.tutorialspoint.com/python/python_classes_objects.htm

In Python, the runtime environment 'knows' which object (i.e. instance of a class) you are using when you call a method or access an attribute. It does this by silently passing a 'self' pointer into the class' code. Therefore all your methods have to have 'self' as their first argument, and you should refer to all attributes as 'self.attribute'. This maintains the scope of the name you've used to be precisely the one which belongs to the object.

class X(Y)

"Make a class named X that is-a Y."

class X(object): def __init__(self, J)

"class X has-a __init__ that takes self and J parameters."

class X(object): def M(self, J)

"class X has-a function named M that takes self and J parameters."

<in both of the above, the class is declared to explicitly inherit from the base 'object' class common to all Python types.>

foo = X()

"Set foo to an instance of class X."

foo.M(J)

"From foo get the M function, and call it with parameters self, J."

foo.K = Q

"From foo get the K attribute and set it to Q."

(http://learnpythonthehardway.org/book/ex41.html)

Other very good resources:

http://www.toptal.com/python/python-class-attributes-an-overly-thorough-guide

http://www.diveintopython.net/object_oriented_framework/class_attributes.html

http://effbot.org/pyfaq/how-can-i-overload-constructors-or-methods-in-python.htm

http://stackoverflow.com/questions/682504/what-is-a-clean-pythonic-way-to-have-multiple-constructors-in-python (also factory methods)

Completeness

Everything that is needed is present

Sufficiency

Everything that is present is needed

Primitiveness

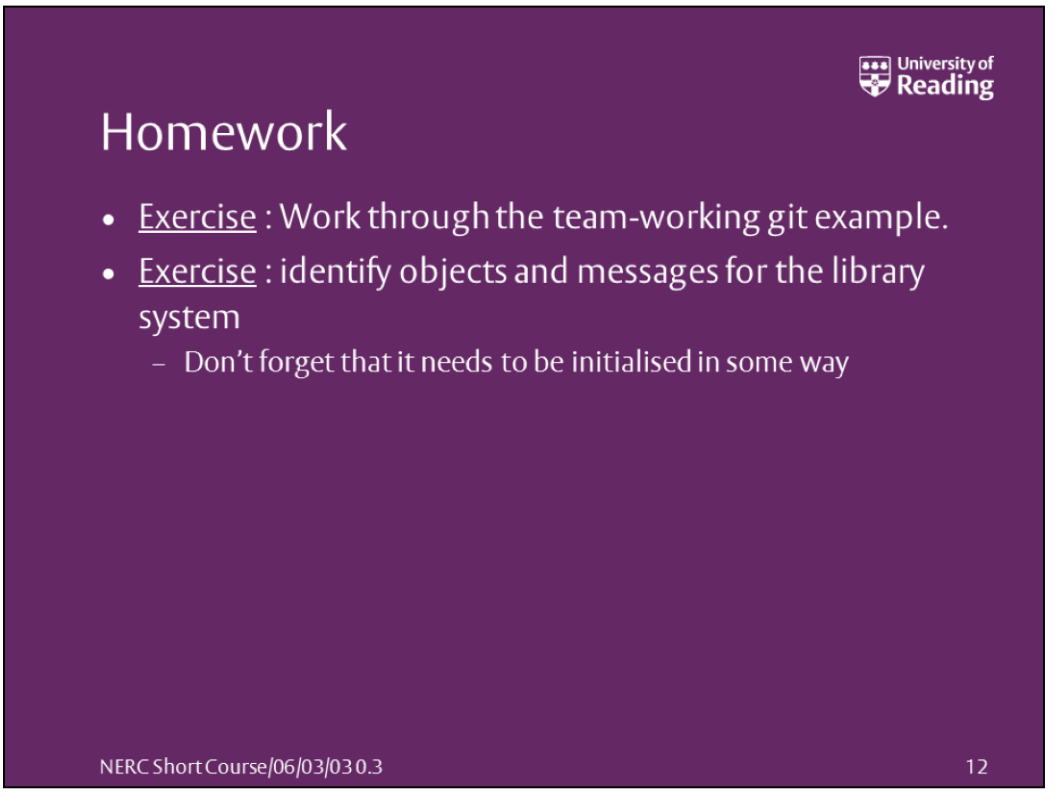
Everything that is present is primitive

High cohesion

Everything that is present is required to be together

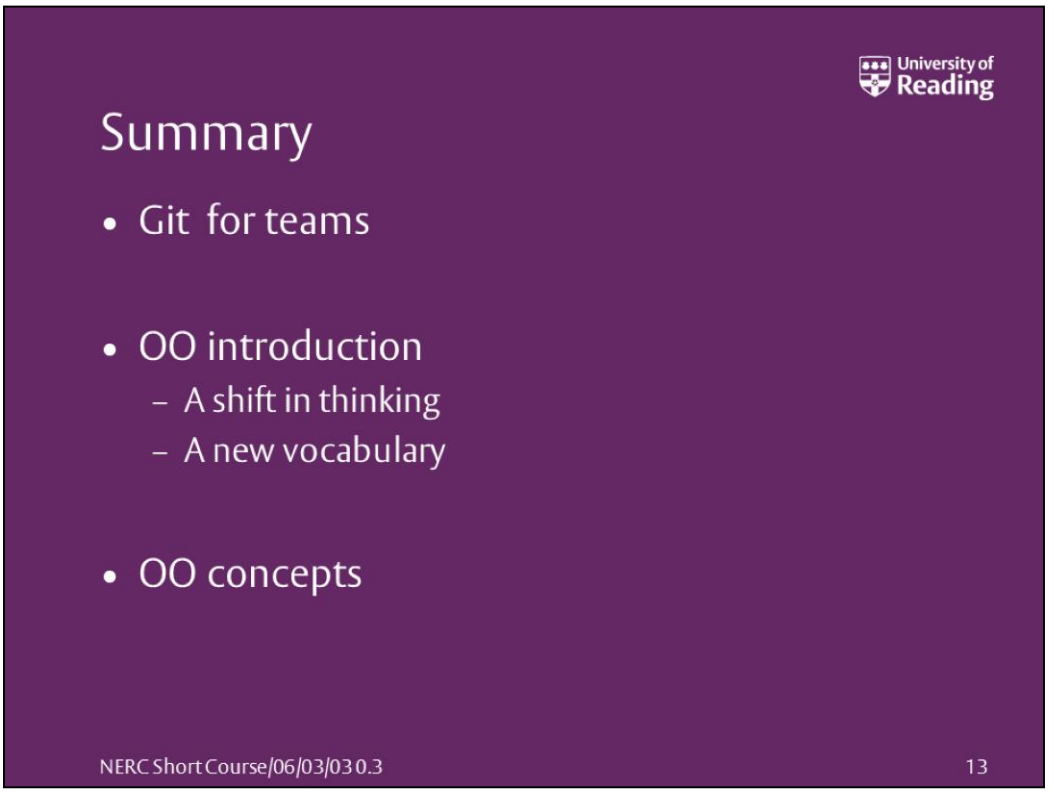
Low coupling

Everything to which we are coupled is required



For the library system, you can assume that you will be provided with a list of book titles in order to populate its catalogue. You should also be able to add a few DVD and journal titles using your classes.

It may be handy to know that a helper class could provide you with incrementing unique numbers to use as identifiers.



Everyone should have drawn a conceptual representation of the Git distributed organisation and hopefully something which follows the steps they've taken…. Compare these with each other.