Week 8 - Coupling and Cohesion

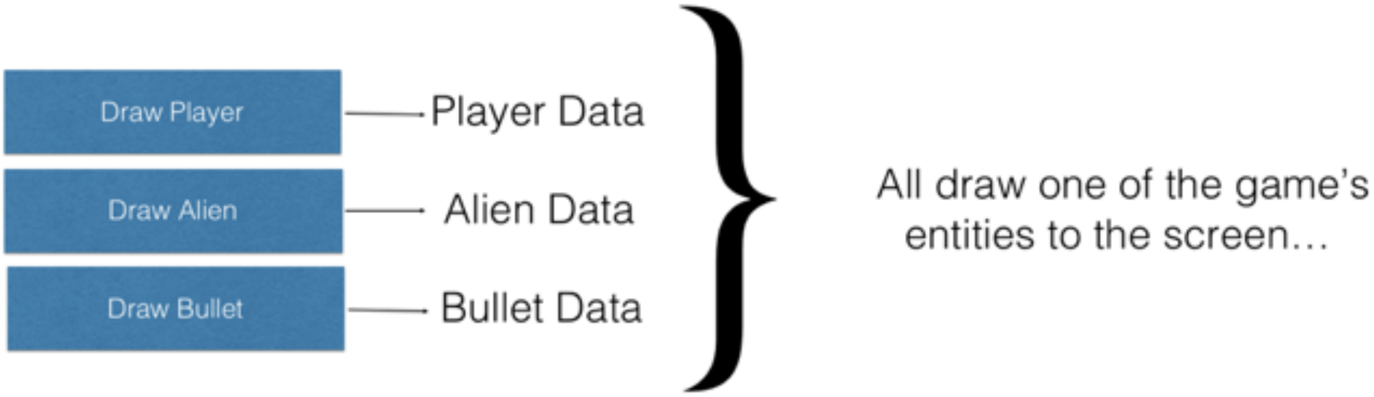
Coupling, Cohesion and Modular Decomposition

In this topic we look at:

- Functional Decomposition
    - Abstraction/re-use
- Cohesion
- Coupling

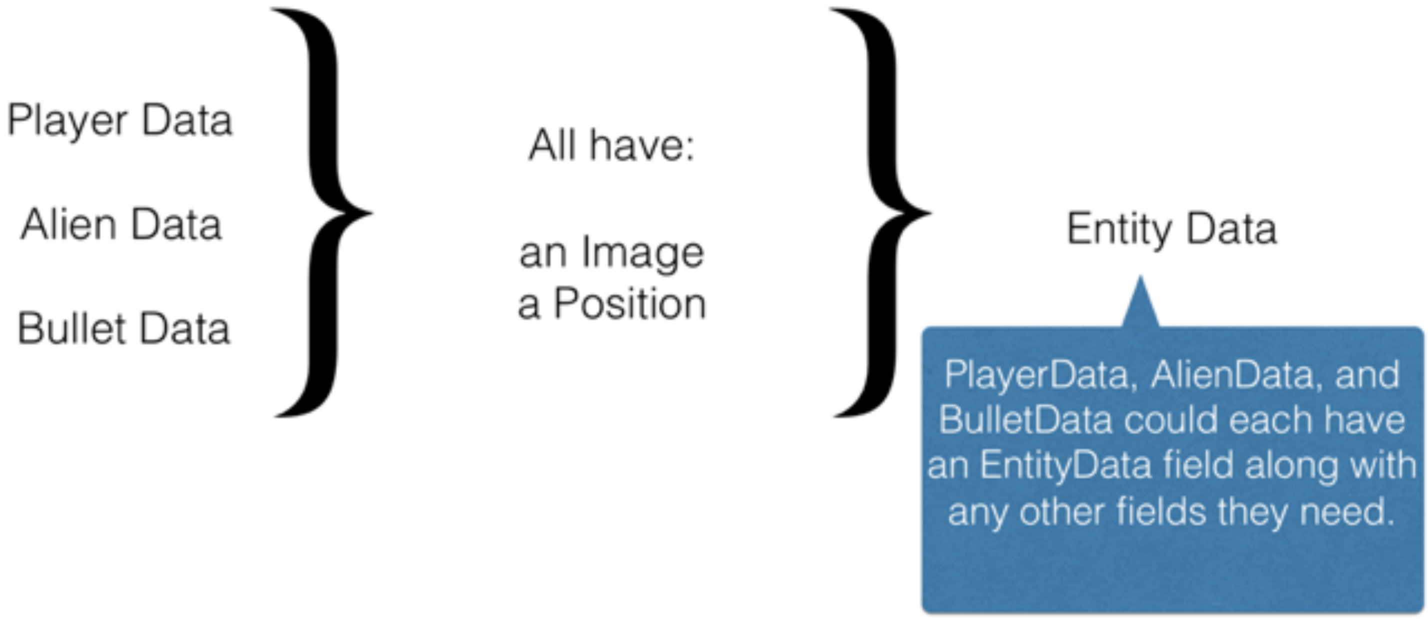
Functional Decomposition – Reuse I

Identify common operations that are implemented in different specific functions/procedures.



Functional Decomposition - Reuse II

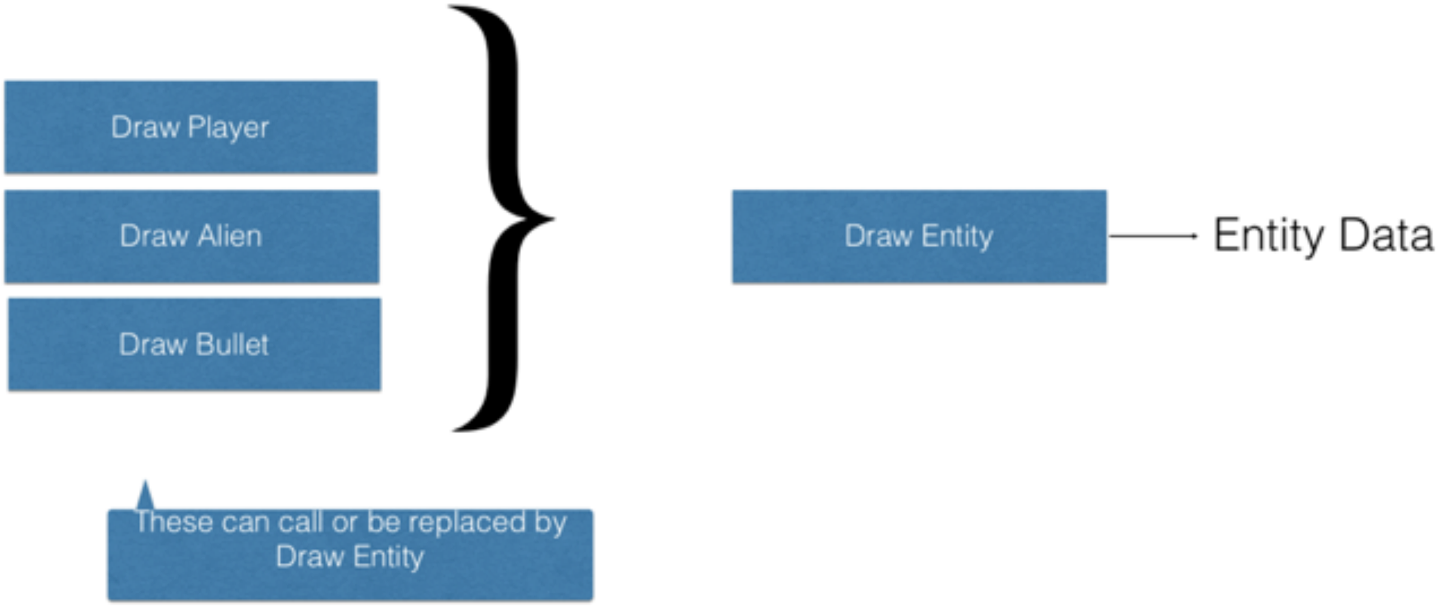
Identify the common data and use this to create a new, more general, record to model this.



PlayerData, AlienData, and BulletData could each have an EntityData field along with any other fields they need.

Functional Decomposition – Reuse III

Recode the specific routines with a new general routine that works on the general data.



Systems Thinking

Various models of systems have been proposed (Jackson 2003):

•Reductionism - the scientific method adopts this approach. It implies that you can study parts in isolation of the whole and in doing so eventually understand the whole.

•Holism - some systems have emergent properties which are only revealed when the system is complete (eg: studying individual termites on their own will not reveal any knowledge about their ability to construct sophisticated termite mounds). Rejects reductionism as useful for complex systems (except at stable levels - see below).

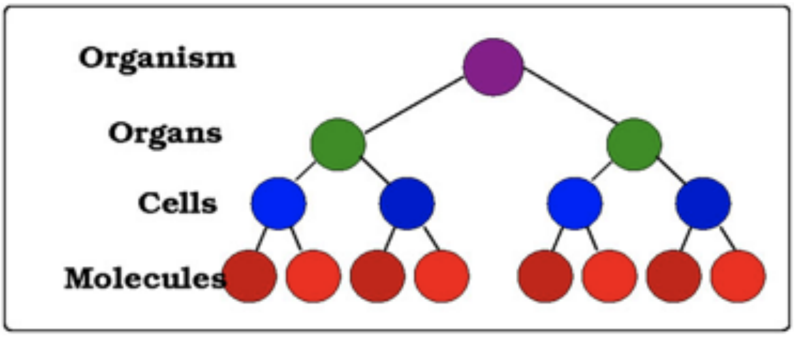
•Ludwig von Bertalanffy added to holism the concept of closed and open systems. A closed system can be considered in isolation from its environment as it has few or no exchanges with it. An an open system, on the other hand, would interact with its environment taking inputs and returning outputs. The separation of the identifiable system from the environment determined the system boundary.

•Norbert Wiener - emphasised control systems with negative and positive feedback and communication. This feedback is used to determine how well the system is achieving its goals and to modify behaviour accordingly.

Holism

More than a sum of parts – at levels in the hierarchy, stable levels of organised complexity arise that have emergent properties.

For example, the organs such as the liver can be considered a system of its own as can a complete animal (organism).

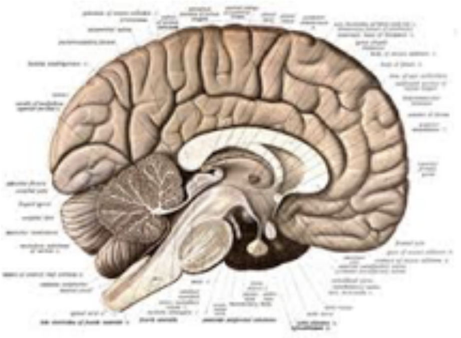


Ludwig von Bertalanffy argued that organisms needed to be studied at levels that were stable (as complex wholes), you would not learn much about animals or their organs by studying their cells alone.

i Jackson, C., 2003 Systems Thinking: Creative Holism for Managers, Wiley.

Reductionism vs Holism

Kaye (1993) describes reductionism as: nothing-but-ism. Eg: a human being is “nothing but” so many litres of water, so many grams of calcium, etc. An example is a human brain, which when looked at as an organ - removed from the body - does not reveal the full range of its capabilities:



Reductionism is analysis (literally: cut up into parts).

Holism, on the other hand, considers systems in their entire complexity (complexity means to plait together).

Another example is a termite mound, studying a single termite does not reveal anything about how termite colonies might work, it is only the whole the reveals the complexity:

Kaye, B., 1993 Chaos and Complexity: Discovering the surprising patterns of Science and Technology, VCH, Weinheim.

Types of Reductionism

Methodological – scientific method, structured programming.

Epistemological (theory) – eg. that new theories reduce older theories to simpler terms (relate to abstraction)

Ontological – the belief that everything can be understood through reductionism.

> See, for example: https://plato.stanford.edu/entries/reduction-biology/

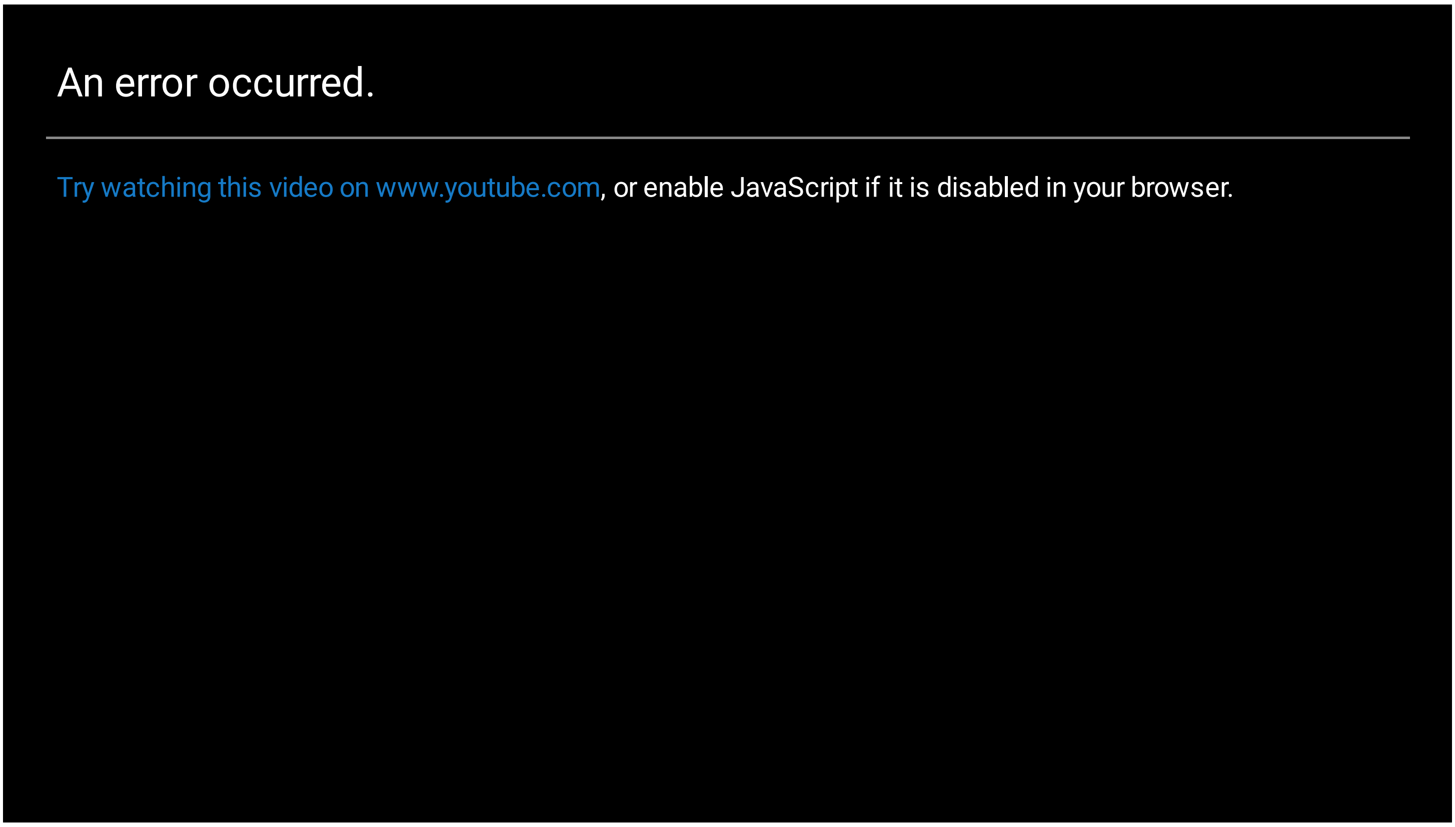
Practical Implications

Holism:

Complex Systems research (eg: study of earth quakes, ant colonies, neural networks)

You can produce models of systems using methodological reductionism, without subscribing to ontological reductionism.

One example is the ArborGames model from the Sante Fe Institute:



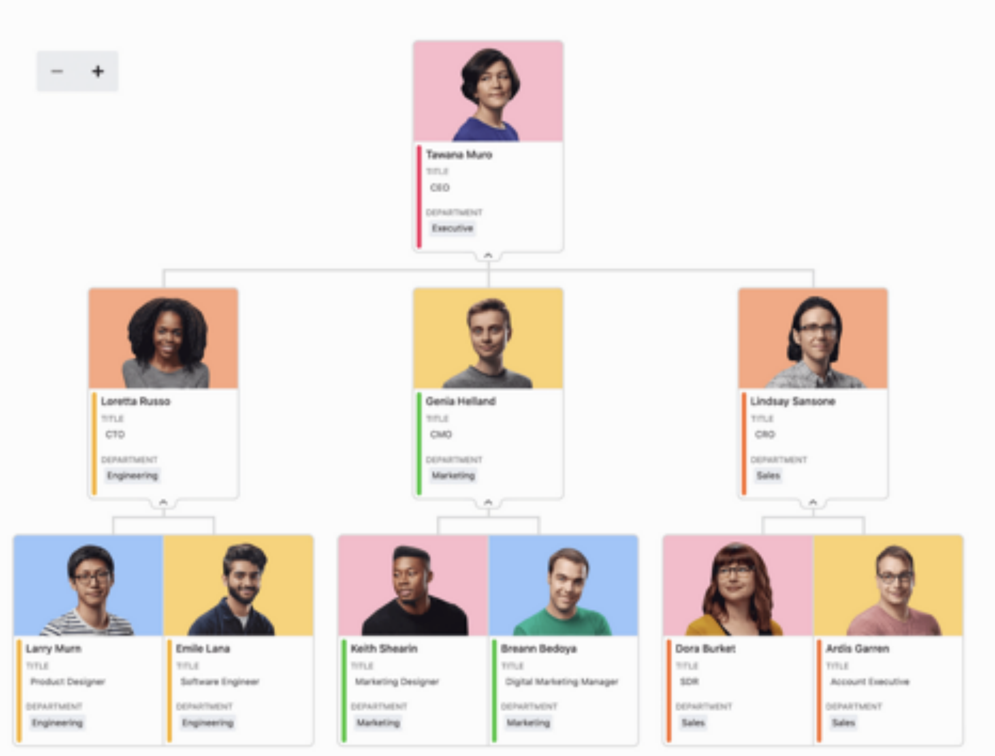
An error occurred.

Try watching this video on www.youtube.com, or enable JavaScript if it is disabled in your browser.

Structure Charts

- Built on a ‘management’ model.
- Reductionist in approach
- Higher level managers, then middle level, then lower level subordinates (workers).

Similar Hierarchical charts: Organisational charts



Tawana Muro
TITLE
CEO
DEPARTMENT
Executive

Loretta Russo
TITLE
CTO
DEPARTMENT
Engineering

Genia Helland
TITLE
CMO
DEPARTMENT
Marketing

Lindsay Sansone
TITLE
CRO
DEPARTMENT
Sales

Larry Murn
TITLE
Product Designer
DEPARTMENT
Engineering

Emile Lana
TITLE
Software Engineer
DEPARTMENT
Engineering

Keith Shearin
TITLE
Marketing Designer
DEPARTMENT
Marketing

Breann Bedoya
TITLE
Digital Marketing Manager
DEPARTMENT
Marketing

Dora Burket
TITLE
SDR
DEPARTMENT
Sales

Ardis Garren
TITLE
Account Executive
DEPARTMENT
Sales

Source: https://try.airtable.com/

Coupling and Cohesion

Remember programs consist of both data and logic.

- Data is organised and passed around according to the principles of coupling.
- Logic (code) is organised according to the principles of cohesion.

One aim of structured design (and OOP) is to reduce the coupling between sections of code which can be logically separated from each other. This has the side-effect of also making code more functionally cohesive and the potential benefits of achieving reuse and avoiding unnecessary code changes later.

- Coupling: how task implementations interact with each other
- Cohesion: how related are task implementations that are grouped together

i See: Robertson, L.A 2014, Students Guide to Program Design, Newnes. Chapt 8. Available online from the Swinburne Library.

Venn Representation of Coupling and Cohesion

Coupling

Reducing coupling means that code that does task *A* and code that does task *B* are not tied too closely together. Loose coupling should allow the code which does *B* to be changed or replaced without having to change the code which does *A*.

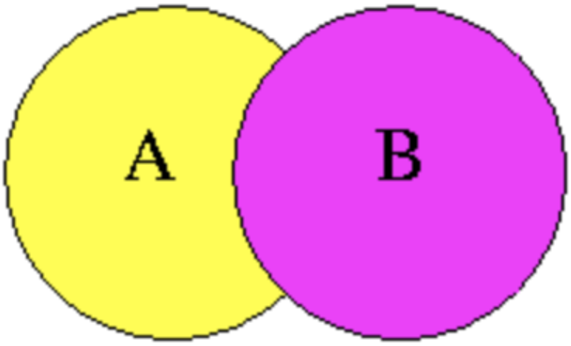
Example of loose coupling: *A* maintains a database of workers. *B* produces paper reports from the database. We should be able to now change *B* so that it produces web-page reports without having to change *A*.

Cohesion

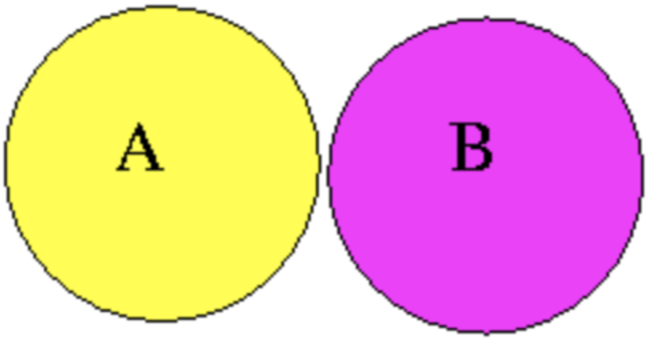
Increasing cohesion means that code that does task *A* should be grouped together and not contain code that does other tasks such as *B*.

Example of low cohesion: System *C* maintains a database, produces reports *and* controls PABX.

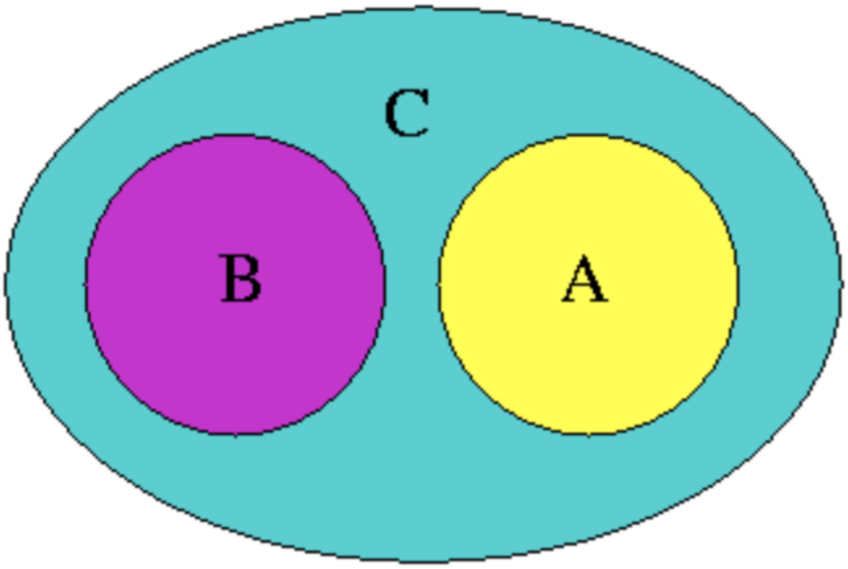
Consider the following representations:



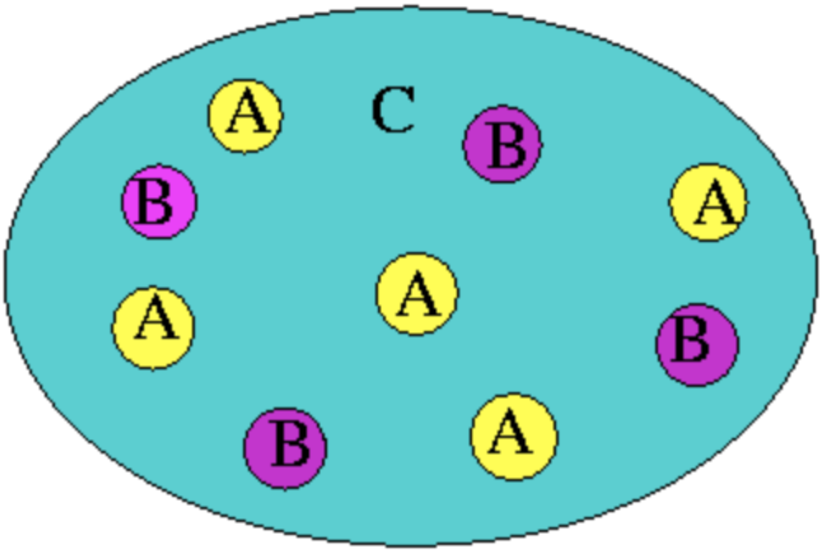
A and B substantially overlap (dependent on each other)



A and B are independent



C does both A and B, but A and B are independent within C



Bits of A and B are scattered throughout C

Note: there may be trade-offs, to reduce coupling lower cohesion may be required. eg. Expert Pattern

Two basic principles:

In relation to code: The Black Box

- Internals are hidden.
- Focus is on how to use rather than how it works

In relation to data: Information Hiding

- Only give access to data that is needed
- Has two aims:
  - Security (i.e prevent unintended or unauthorized access)
  - Reduce complexity – only think about what you need here and now. Part of decomposition.

Cohesion

Aim:

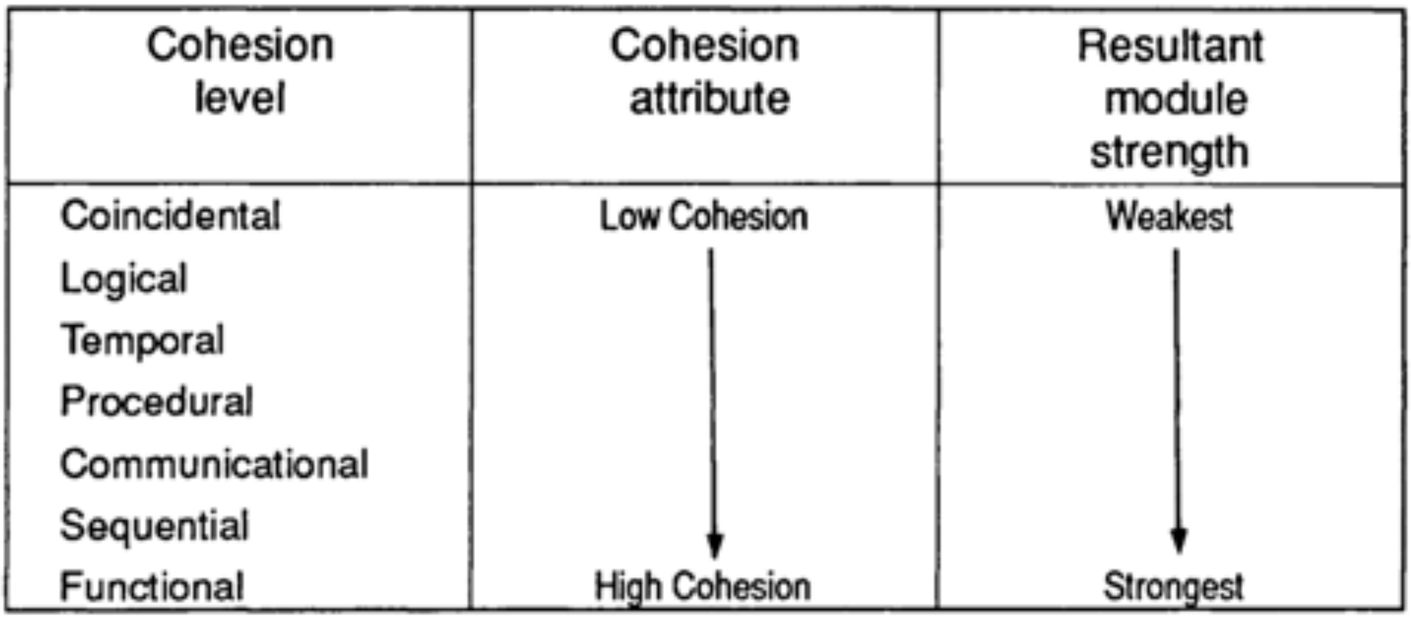
Create modules whose elements are strongly and genuinely related to one another.

Remember that coupling was how the modules are separated from each other, these two ways of partitioning systems are related to each other. High cohesion can lead to low coupling in many cases.

Types of Cohesion

- Functional cohesion: all elements are related directly to a single task.
- Sequential cohesion: output of one element serves as input to next.
- Communicational cohesion: two elements are in the same module because they share the same data.
- Procedural cohesion: elements involve unrelated activities but control flows from one to the next.
- Temporal cohesion: grouping together elements based on the fact they all occur at the same time
- Logical cohesion: this includes a number of elements of which are generally related but of which only a sub-set are likely to be used at any one time.
- Co-incidental cohesion: a module whose elements are in no way related.

Cohesion Levels



Cohesion level	Cohesion attribute	Resultant module strength
Coincidental	Low Cohesion	Weakest
Logical		
Temporal		
Procedural		
Communicational		
Sequential		
Functional	High Cohesion	Strongest

The top one is generally considered least desirable, the bottom most. But what you choose is up to you as the designer - and in specific situations some of the lower cohesion types might in fact be the most appropriate to the overall design.

Functional cohesion

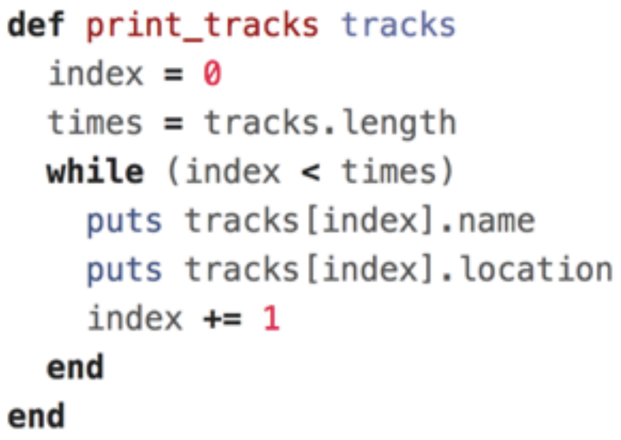
Functional cohesion: all elements are related directly to a single task.

A non-functionally cohesive module can be detected by names such as:

`calculate_tax_and_deduct_medicare_payment()`

The presence of ***and*** in a well named method indicates the lack of cohesion.

A functionally cohesive example is:



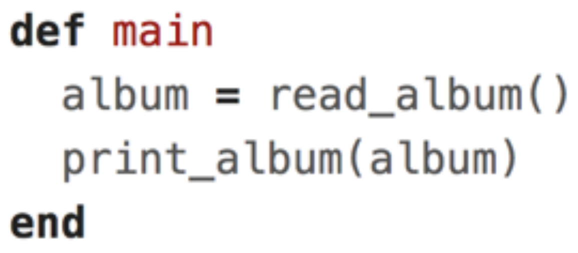
```
def print_tracks tracks
  index = 0
  times = tracks.length
  while (index < times)
    puts tracks[index].name
    puts tracks[index].location
    index += 1
  end
end
```

Sequential cohesion

Sequential cohesion : output of one element serves as input to next

eg: generate report lines then print report.

Or read albums, then display them:

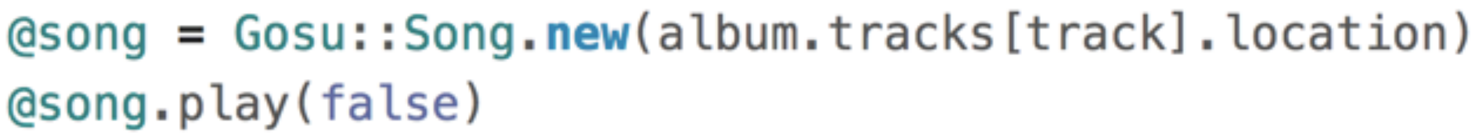


```
def main
  album = read_album()
  print_album(album)
end
```

Quiz: Sequential Cohesion

Question

Is the following code sequentially cohesive?



```
@song = Gosu::Song.new(album.tracks[track].location)
@song.play(false)
```

○ True

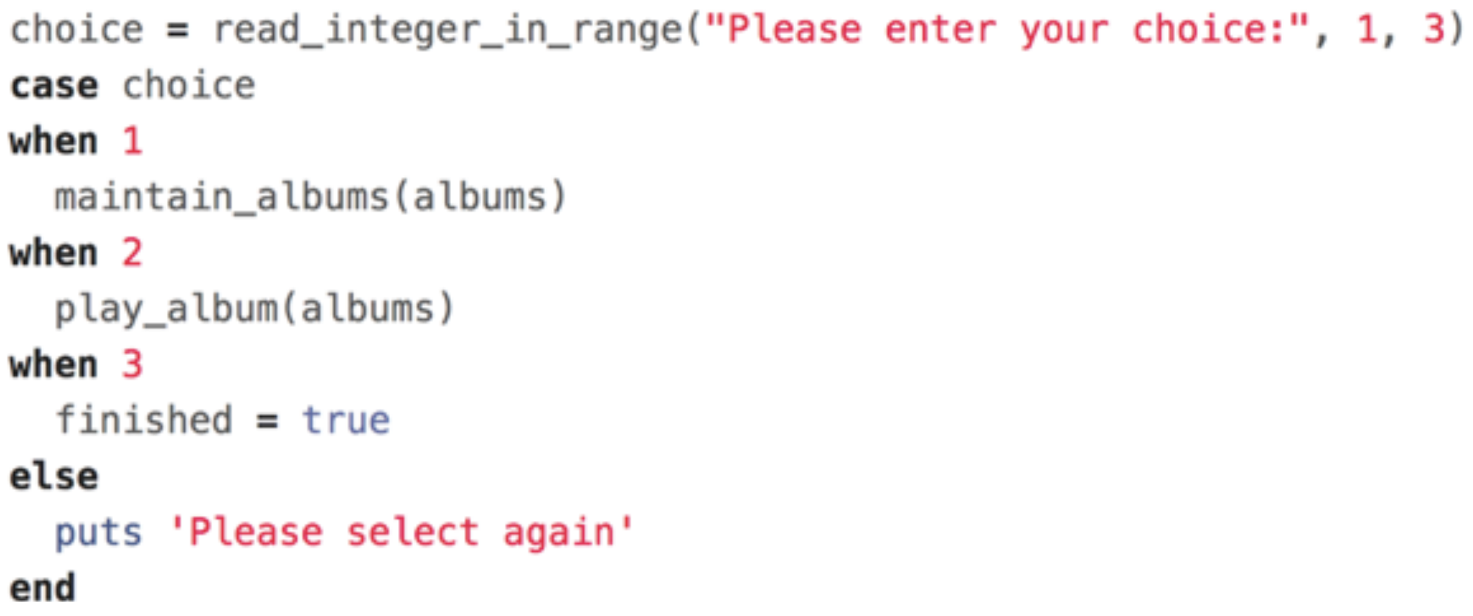
○ False

Communicational cohesion

Communicational cohesion: two elements are in the same module because they share the same data.

eg: find title of book and find author of book (may not be executed in order).

Or:



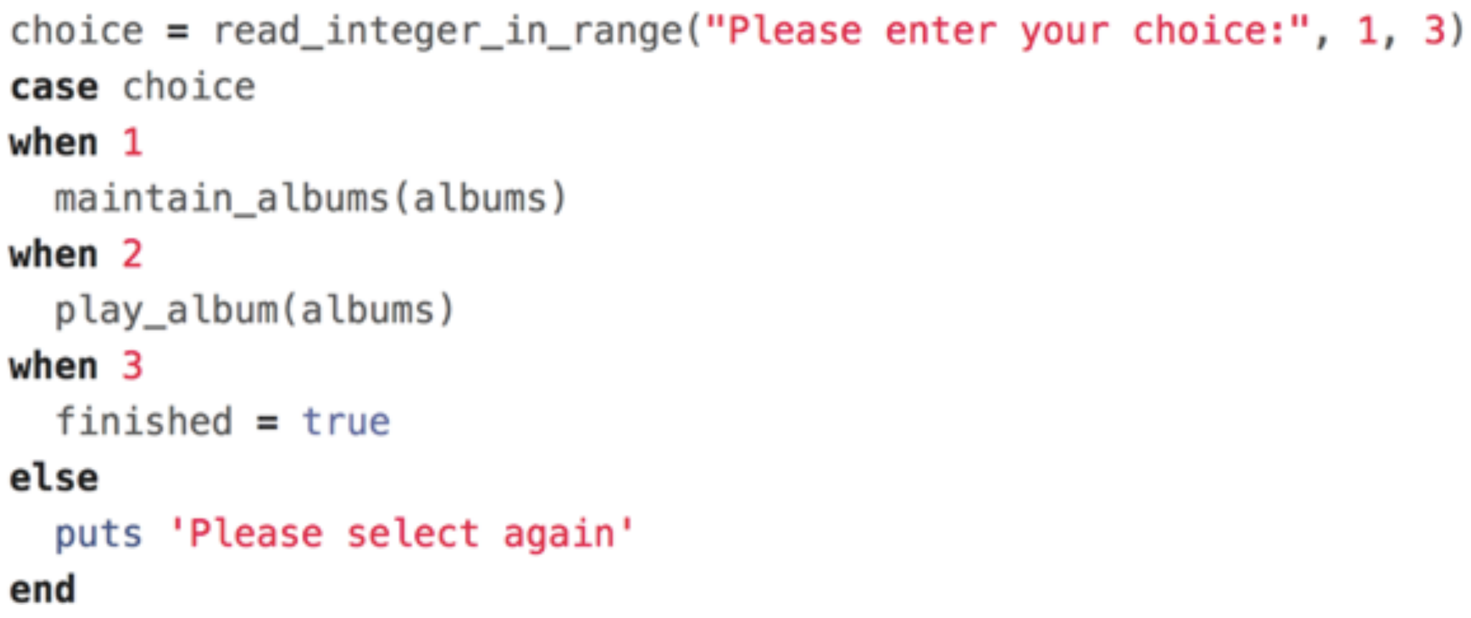
```
choice = read_integer_in_range("Please enter your choice:", 1, 3)
case choice
when 1
  maintain_albums(albums)
when 2
  play_album(albums)
when 3
  finished = true
else
  puts 'Please select again'
end
```

What data is shared between modules in the above?

Quiz: Communicational Cohesion

Question

What data is shared between modules in the code?



```
choice = read_integer_in_range("Please enter your choice:", 1, 3)
case choice
when 1
  maintain_albums(albums)
when 2
  play_album(albums)
when 3
  finished = true
else
  puts 'Please select again'
end
```

○ Choice

○ finished

○ albums

○ 1 and 3

Procedural cohesion

Procedural cohesion: elements involve unrelated activities but control flows from one to the next and there is a general notion of sequence.

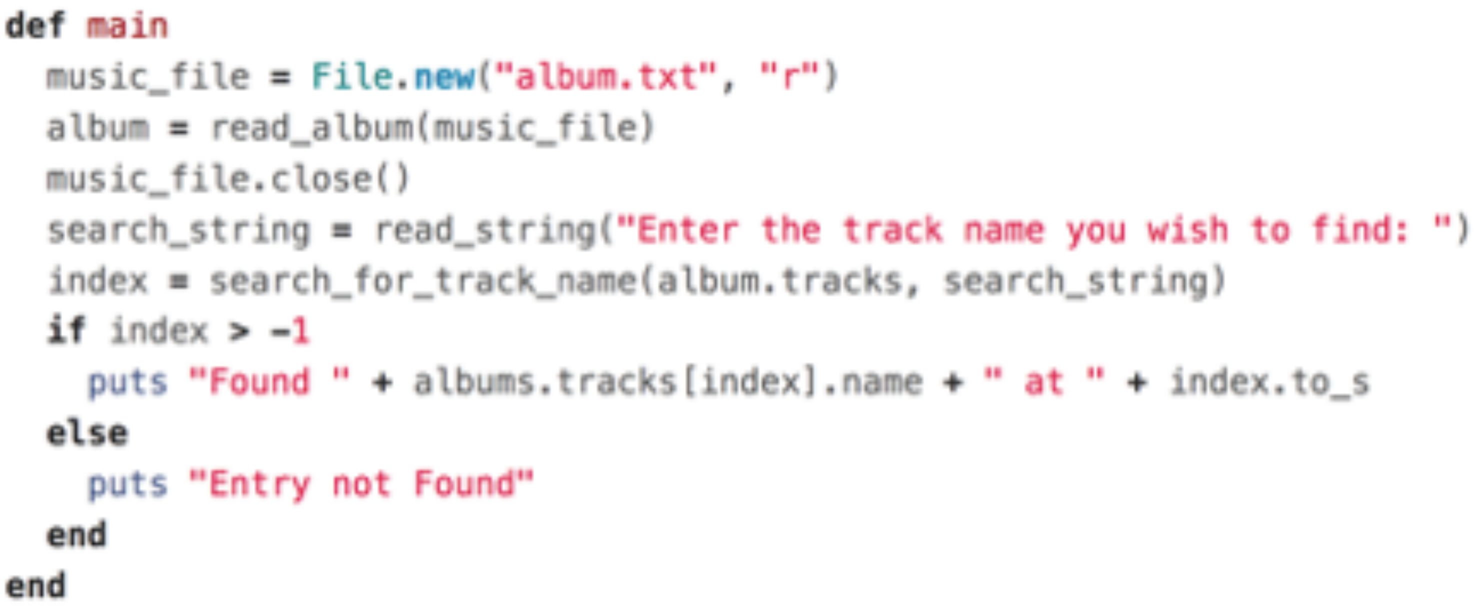
eg: have shower, get dressed, make breakfast, read paper

The order of some of these could be swapped eg: read paper, make breakfast.

They are together because in this particular instance of their use some of the tasks occur in this order.

Quiz: Procedural Cohesion

Question



```
def main
  music_file = File.new("album.txt", "r")
  album = read_album(music_file)
  music_file.close()
  search_string = read_string("Enter the track name you wish to find: ")
  index = search_for_track_name(album.tracks, search_string)
  if index > -1
    puts "Found " + albums.tracks[index].name + " at " + index.to_s
  else
    puts "Entry not Found"
  end
end
```

○ The code is procedurally cohesive because there is a general notion of sequence.

○ The code is procedurally cohesive because all the tasks are related

○ The code is not procedurally cohesive

Temporal Cohesion

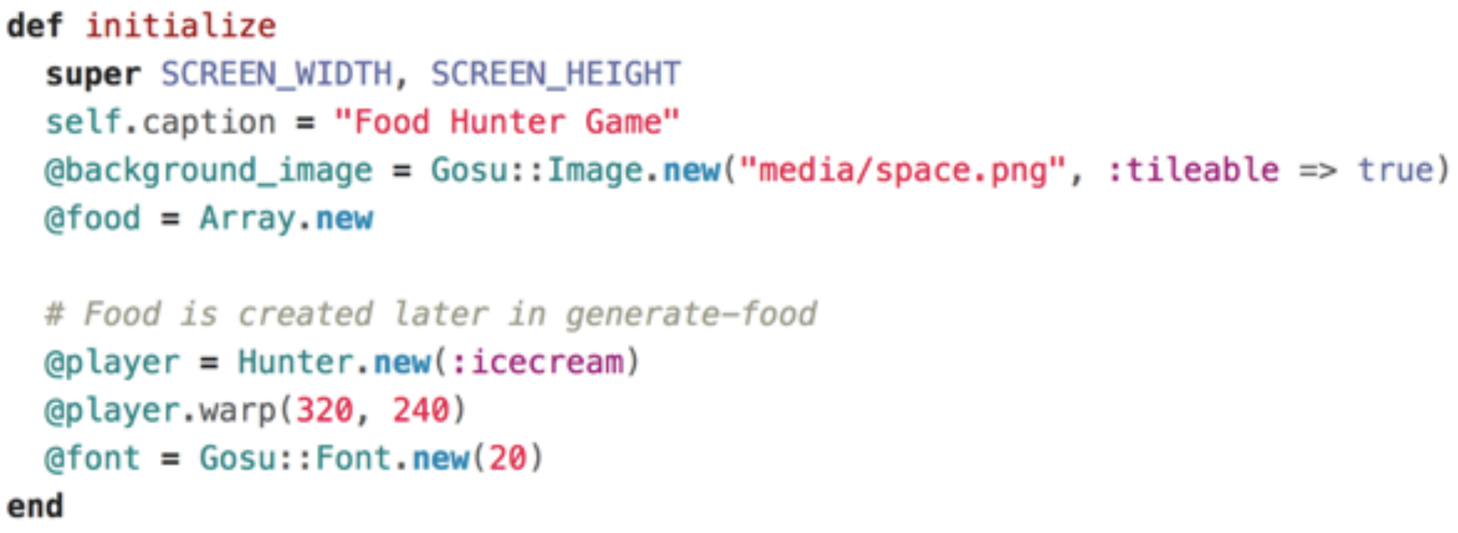
Temporal cohesion: grouping together elements based on the fact they all occur at the same time.

eg: print monthly report and create monthly back-up of database.

Quiz: Temporal Cohesion

Question

Which statement best explains why the following code exhibits temporal cohesion?



```
def initialize
  super SCREEN_WIDTH, SCREEN_HEIGHT
  self.caption = "Food Hunter Game"
  @background_image = Gosu::Image.new("media/space.png", :tileable => true)
  @food = Array.new

  # Food is created later in generate-food
  @player = Hunter.new(:icecream)
  @player.warp(320, 240)
  @font = Gosu::Font.new(20)
end
```

○ Because it is an `initialize()` procedure that is called when records/instances are created.

○ Because it has a number of tasks grouped together only because they occur at the same time.

○ Because it does not demonstrate any of the other types of cohesion.

○ Because it sets up the graphical elements on the screen.

Logical Cohesion

Logical cohesion: this includes a number of elements of which are generally related (i.e of the same class of task) but of which only a sub-set are likely to be used at any one time.

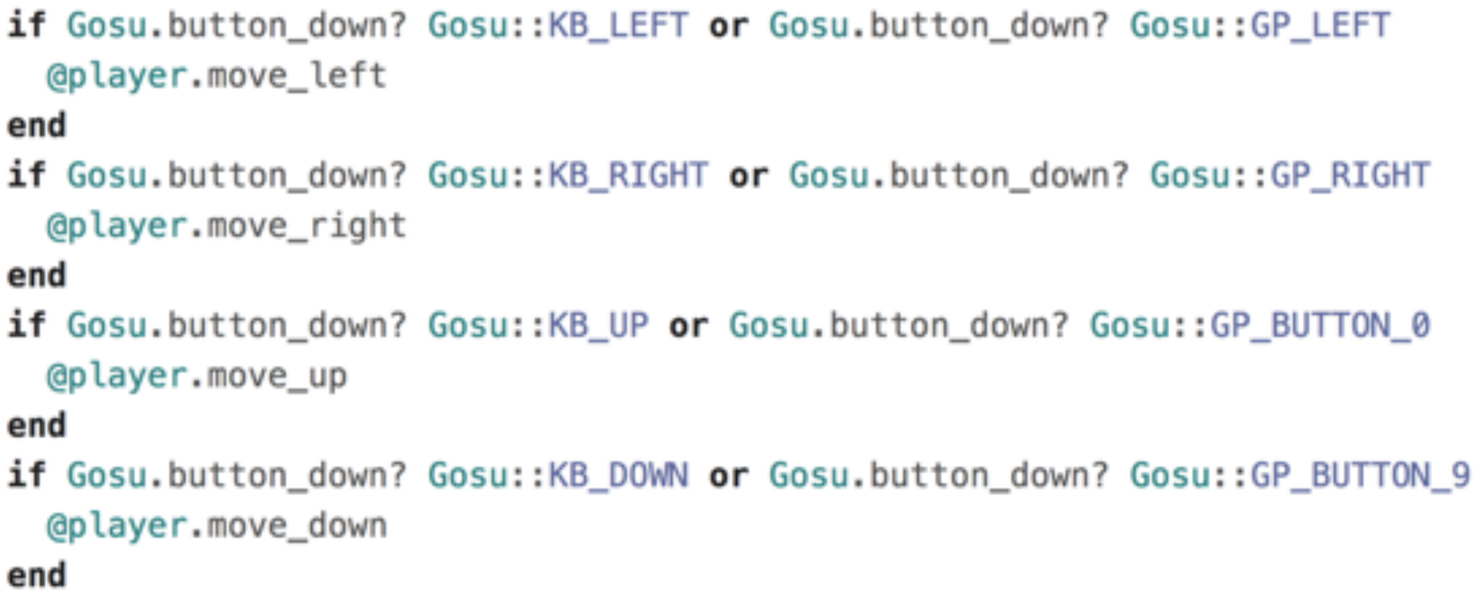
In other words, we pick out the bits we want when we come to use it each time.

Such modules typically have complex interfaces (for switching parameters). Typically they are created to take advantage of shared data structures or code.

Quiz: Logical Cohesion

Question

The following code is logically cohesive - i.e the tasks are put together simply because they share some common data. What is that shared data?



```
if Gosu.button_down? Gosu::KB_LEFT or Gosu.button_down? Gosu::GP_LEFT
  @player.move_left
end
if Gosu.button_down? Gosu::KB_RIGHT or Gosu.button_down? Gosu::GP_RIGHT
  @player.move_right
end
if Gosu.button_down? Gosu::KB_UP or Gosu.button_down? Gosu::GP_BUTTON_0
  @player.move_up
end
if Gosu.button_down? Gosu::KB_DOWN or Gosu.button_down? Gosu::GP_BUTTON_9
  @player.move_down
end
```

○ `@player`

○ `Gosu::button.down?`

○ The Gosu constants

○ The `player.move` procedures/methods (eg: `@player.move_up`)

○ All the above

○ A and B

Co-incidental Cohesion

Co-incidental cohesion: a module whose elements are in no way related.

Similar to a logically cohesive one but with no shared data structures or code. Eg: command line utilities.

- **ls** list files
- **grep** select lines which match search criteria
- **sort** sort the input (-r for reverse order)
- **less** display one page at a time

Eg: use piping (|) to redirect output from one utility to the next.

To try this click on the Terminal window and type (or copy) in the following command:

```
ls | grep file | sort -r | less
```

Coupling

Aim:

Make modules as independent as possible (i.e low coupling)

Remember that cohesion was how modules (functions/procedures) were created by combining code.

These two ways of partitioning systems are related to each other.

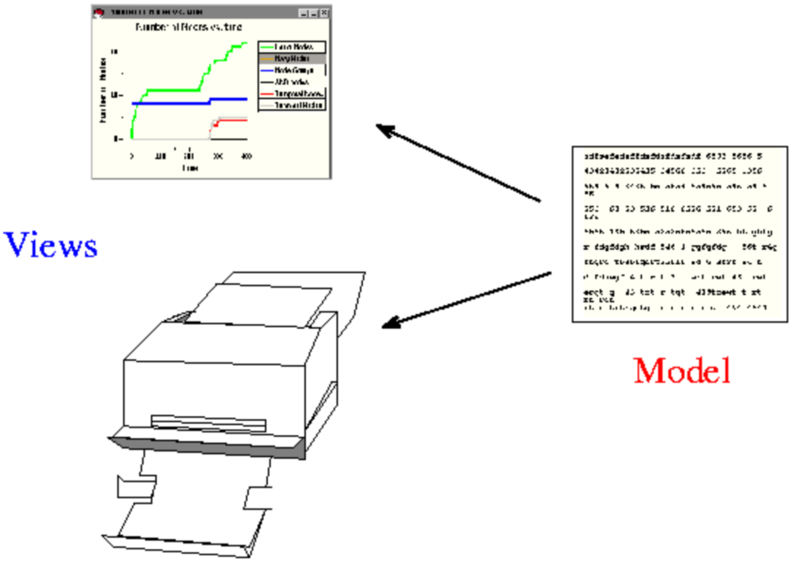
High cohesion can lead to low coupling in many cases.

One design pattern (common design approach) to the problem of coupling is the *Model-View Separation* or *Observer* pattern.

The idea is that the **model** (eg. database system) is separated from the **view** (eg. code which produces either paper or web-based reports).

By analogy, we may have a file which is an Excel file, which we can look at as either a spreadsheet, a graph, or as text. The *model* is the physical text file (or more accurately: the data and data structures), the *view* is the presentation of that file to you or some other system.

The code for model and view should be as independent as possible, we do not want to create a separate file or separate class for each possible view. So changing the view (eg. new graph classes or graphs) should not require changes to the model (eg. model classes or file format).



The Model-View (observer) pattern is often also referred to as **Publisher-Subscriber**.

Why low coupling?

Eliminate unnecessary relationships;

Reduce the number of necessary relationships easing the "tightness" of necessary relationships.

Benefits:

Fewer connections between modules means a problem in one module is less likely to create a problem in another.

Changes to one module are also less likely to require changes to another while maintaining one module, we don't want to worry about the implementation of other modules (i.e simplify understanding).

Principles of coupling I

Create narrow (not broad) connections: modules with lots of connections have a broader interface.

Connections in this sense means the number of calls between each.

Principles of Coupling II

Create narrow (not broad) connections: modules with lots of connections have a broader interface. Connections in this sense means the number of calls between each.

Create direct connections: avoid indirection. Eg. do not bundle data together in such a way that the user (developer) must follow lots of links to work out what the data consists of.

Principles of Coupling III

Create narrow (not broad) connections: modules with lots of connections have a broader interface. Connections in this sense means the number of calls between each.

Create direct connections: avoid indirection. Eg. do not bundle data together in such a way that the user (developer) must follow lots of links to work out what the data consists of.

Create local connections: all information necessary to understand the connection is located together (i.e as parameters to a method call). An example of remote (non-local) information is global data.

Principles of Coupling IV

Create narrow (not broad) connections: modules with lots of connections have a broader interface. Connections in this sense means the number of calls between each.

Create direct connections: avoid indirection. Eg. do not bundle data together in such a way that the user (developer) must follow lots of links to work out what the data consists of.

Create local connections: all information necessary to understand the connection is located together (i.e as parameters to a method call). An example of remote (non-local) information is global data.

Create obvious (not obscure) connections: do not do things one way when they are normally done another. eg. module communicates with another module by changing shared (or global) data, represent phone numbers using ASCII not hexadecimal encodings.

Principles of Coupling V

Create narrow (not broad) connections: modules with lots of connections have a broader interface. Connections in this sense means the number of calls between each.

Create direct connections: avoid indirection. Eg. do not bundle data together in such a way that the user (developer) must follow lots of links to work out what the data consists of.

Create local connections: all information necessary to understand the connection is located together (i.e as parameters to a method call). An example of remote (non-local) information is global data.

Create obvious (not obscure) connections: do not do things one way when they are normally done another. eg. module communicates with another module by changing shared (or global) data, represent phone numbers using ASCII not hexadecimal encodings.

Create Flexible (not rigid) connections: a module gets its data directly from a source specific to a particular task (eg. a remote server). If we wish to use this module for something else (i.e a local source) we cannot.

Design Trade-Offs

Sometimes trade-offs have to be made.

To reduce coupling lower cohesion may be required. eg. Expert Pattern

These are design decisions that depend on the problem, but one approach is to design based on the priorities in the previous slides.

> Note that modules can also be coupled by their shared use of database records. One module may break the integrity rules of the database. Since this may not be detected for some time, to allow the tracking of such a problem, each time a module updates a record also update a record header with audit information.

Bad Coupling

Some examples of dangerous coupling (there are many more):

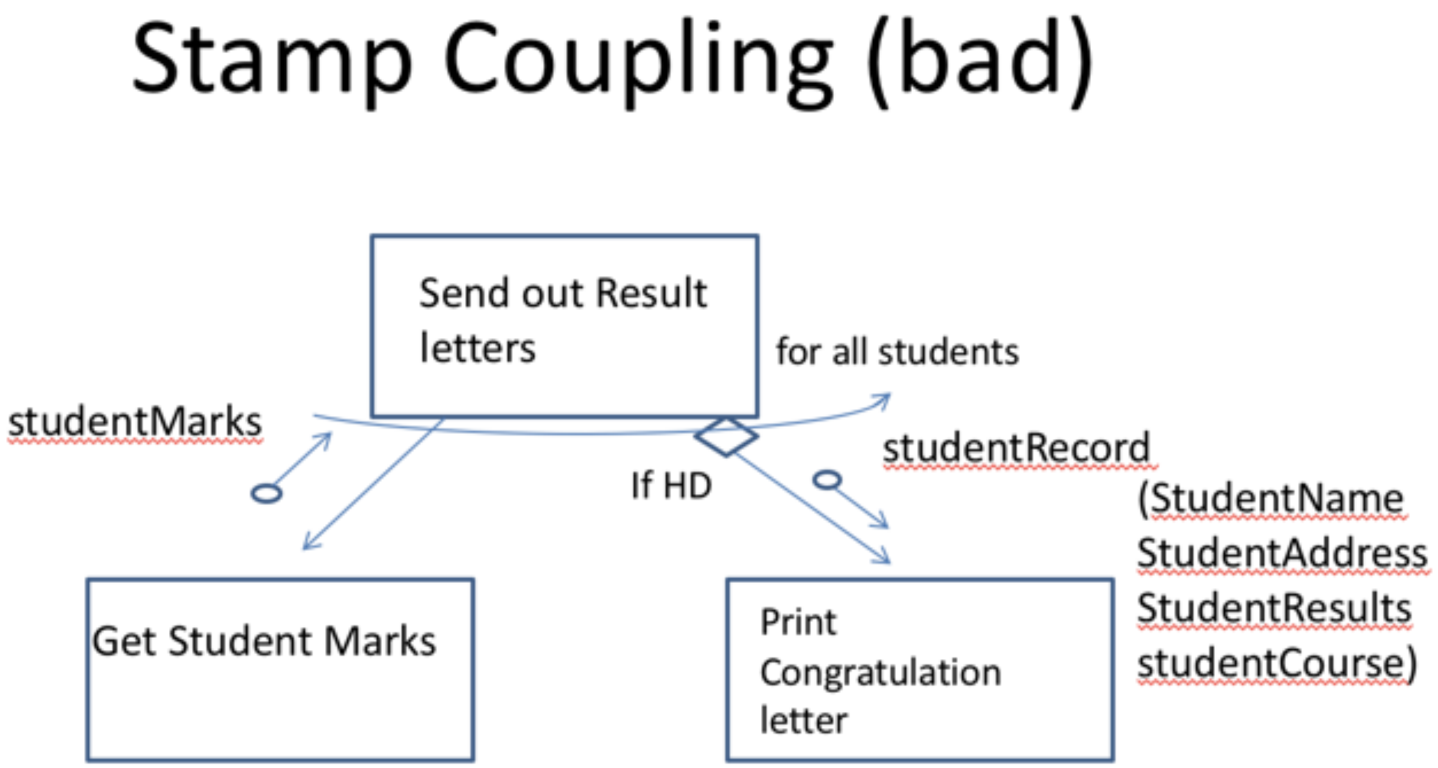
- **Stamp coupling**: Passing records to an object that requires only a couple of fields within the record (i.e maintain information hiding).
- **Control coupling**: Passing in data that controls the behaviour of the called module.
- **Hybrid coupling**: having some values of a parameter indicate values to be processed and other values used to control the behaviour of the called module.

Quiz: Stamp coupling

Question

Which of the following **best** describes the stamp coupling the example below.

This structure chart is for a program that prints out a letter for each student in the Introduction Programming course who achieved a HD.



Stamp Coupling (bad)

Send out Result letters

for all students

studentMarks

If HD

studentRecord

(StudentName
StudentAddress
StudentResults
studentCourse)

Get Student Marks

Print Congratulation letter

○ The module 'Print Congratulation Letter' does not require any fields from the studentRecord.

○ The module 'Print Congratulation Letter' requires some fields from the student record, but only one result, not all the results.

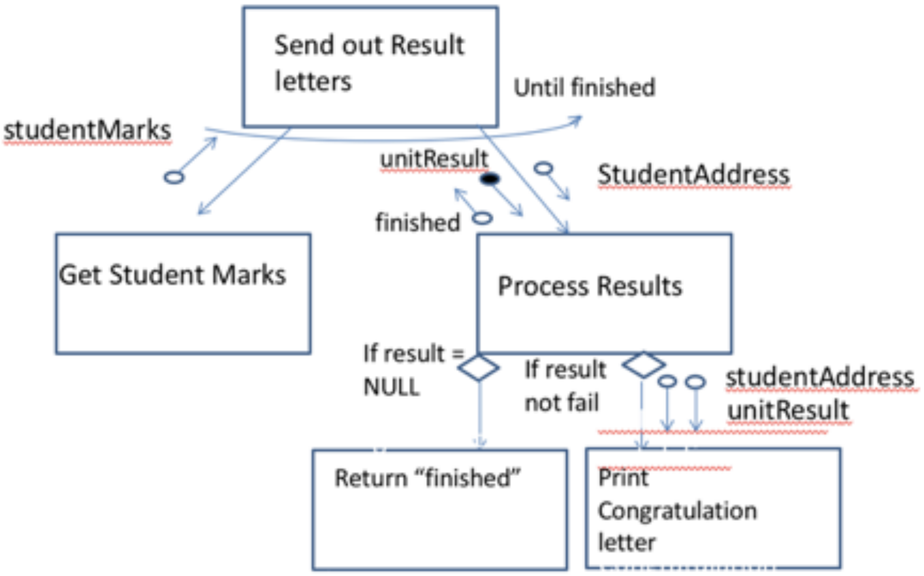
○ The module 'Print Congratulation Letter' requires only the student name and address.

Quiz: Control Coupling

Question

In the following structure chart the parameter `unitResult` is used for both data and control (so it is a hybrid). Which statement below best captures the worst design problems in this example?

Hybrid Coupling (bad)



□ Hybrid coupling arrows should not go down as this shifts decisions from the manager to the worker module.

□ Ideally variables should not be used for both control and data.

□ The finished state should be determined in the manager rather than tested in the worker.

□ A and C

□ All of the above.

Summary

We have looked at structured design principles.

These include:

- Functional decomposition
- Cohesion
- Coupling (for data passing between modules)

The aim is create code that is both easily maintainable and potentially reusable.