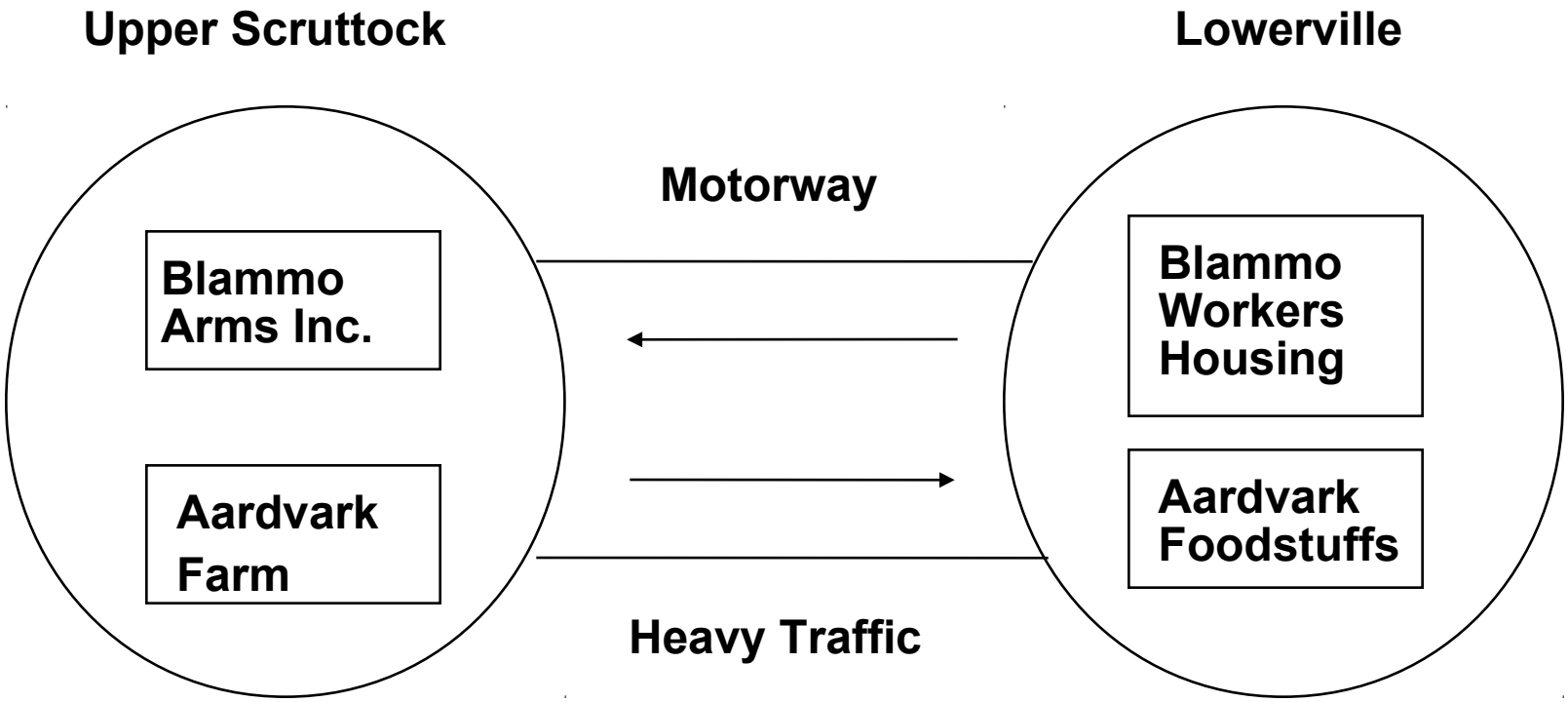
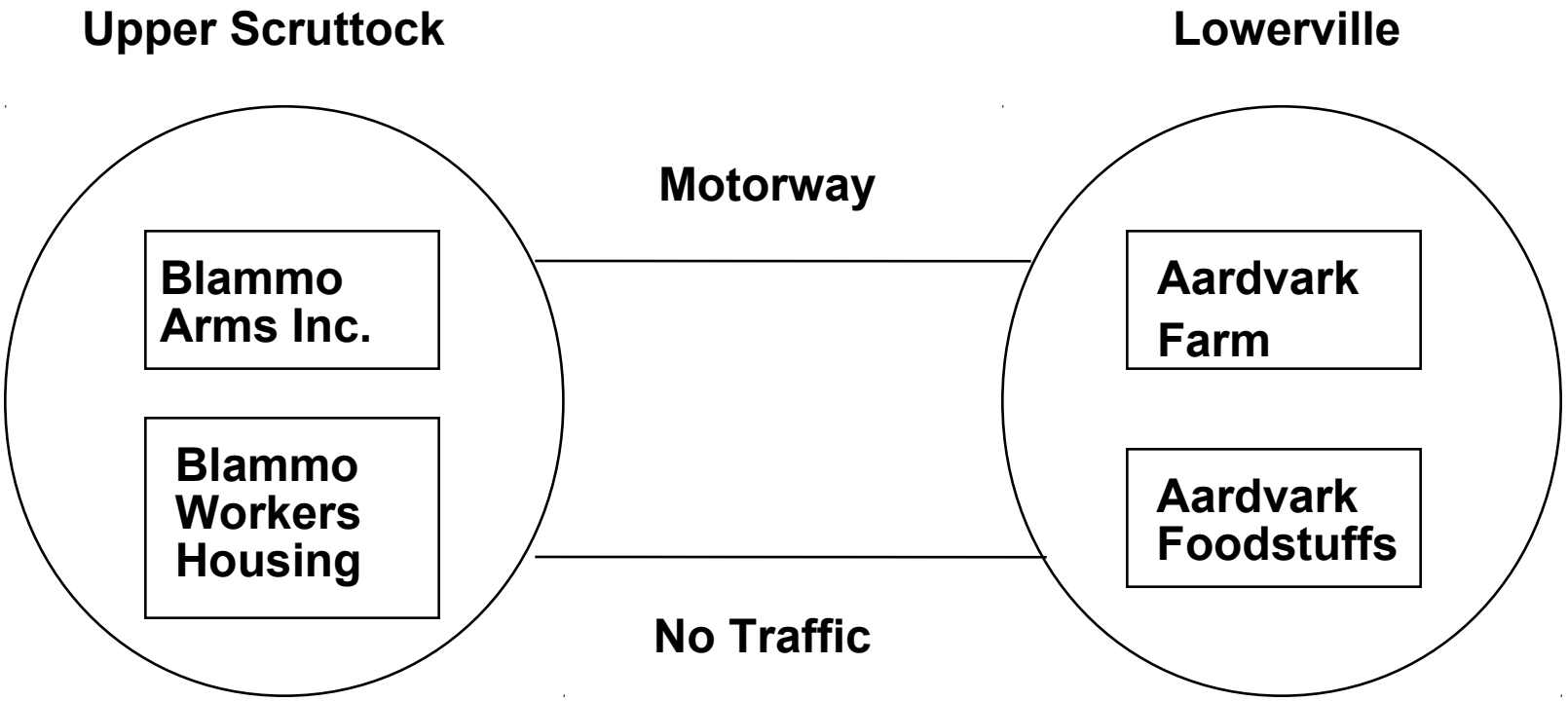
Good Design

- Many of these issues true of (come from) structured and OO approaches.
  - In addition also been applied to other areas, e.g., coupling in roles.
- Mainly heuristics (though suggests measures) for:
  - Cohesion
  - Coupling
  - Connascence

Cohesion and Coupling: Less than ideal



A Better Solution. More Cohesive and less Coupled



Cohesion

- Strength of functional ‘relatedness’ of activities.
  - An activity is an instruction, or a group of instructions, a data definition or a call to other services.
  - Designers should create strong, highly cohesive services whose elements are strongly and genuinely related to each other.

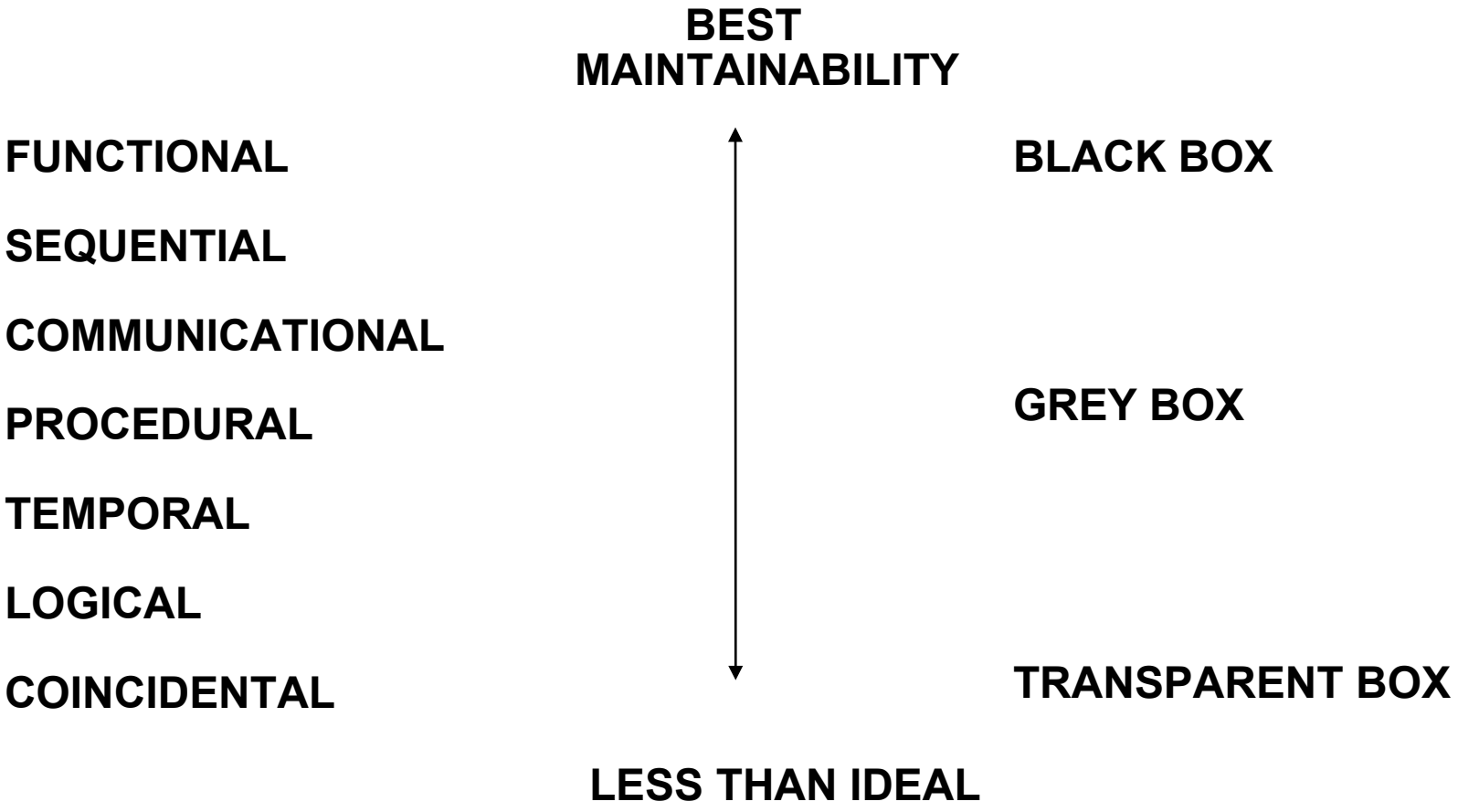
Cohesion in Objects

- How well the developer has partitioned a system into objects.
- Making sure objects are strong cohesive abstractions may also minimise coupling.
  - Cohesive objects are built through anthropomorphism techniques.
  - Most existing cohesion theory is based upon functional paradigms.

Service Cohesion

- The principles of cohesion are typically applied at the individual service level.
- Develop object services that fulfill a well defined role.
  - E.g., a service that calculates interest on a given sum is good.
  - A service that calculates interest, washes milk bottles and parks a hard disk all at once is not.

Levels of Cohesion

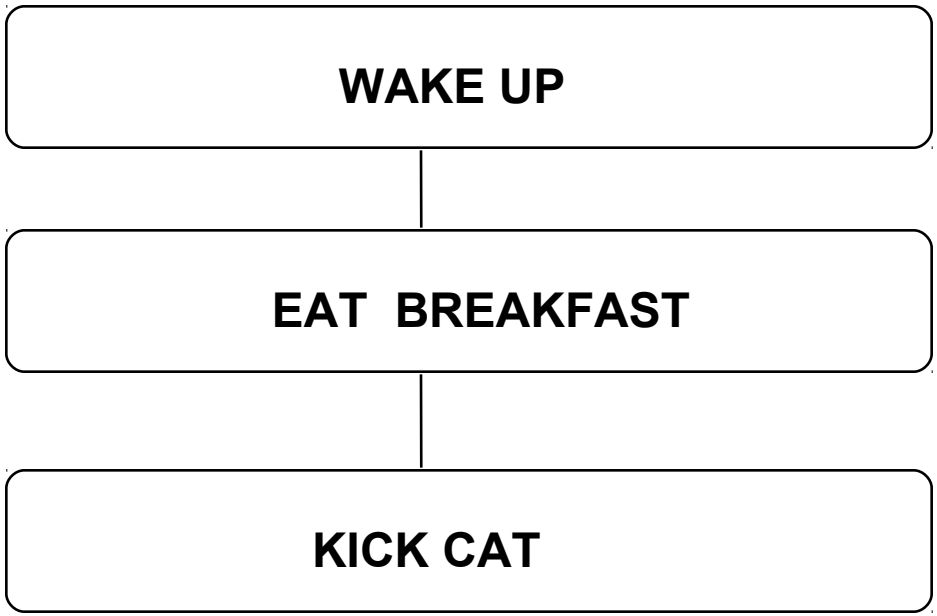


Functionally Cohesive

- Elements contribute to the execution of one problem related task
  - E.g., Start_Pump, Make_Reservation, Eat_Fish etc.
- Crisp functional abstraction that has a well defined role.
- Easiest and to maintain ...
  - however, it is not always possible to define services that fulfill a single task or role.

Sequentially Cohesive

- Objects encapsulate activities where the output from one activity is the input to the next.



Functionally Cohesive Services

- A sequentially cohesive service reduces coupling by encapsulating related functionality.
- Arguably as maintainable as a functionally cohesive object.
- However, may not be as good for reuse as it may contain activities that will not generally be useful together.

Communicationally Cohesive

- Elements contribute to activities that use the same input or output data.
- E.g., given an ISBN number as input to a service...
  - FIND TITLE OF BOOK
  - FIND PRICE OF BOOK
  - FIND PUBLISHER OF BOOK

Comparing Cohesion

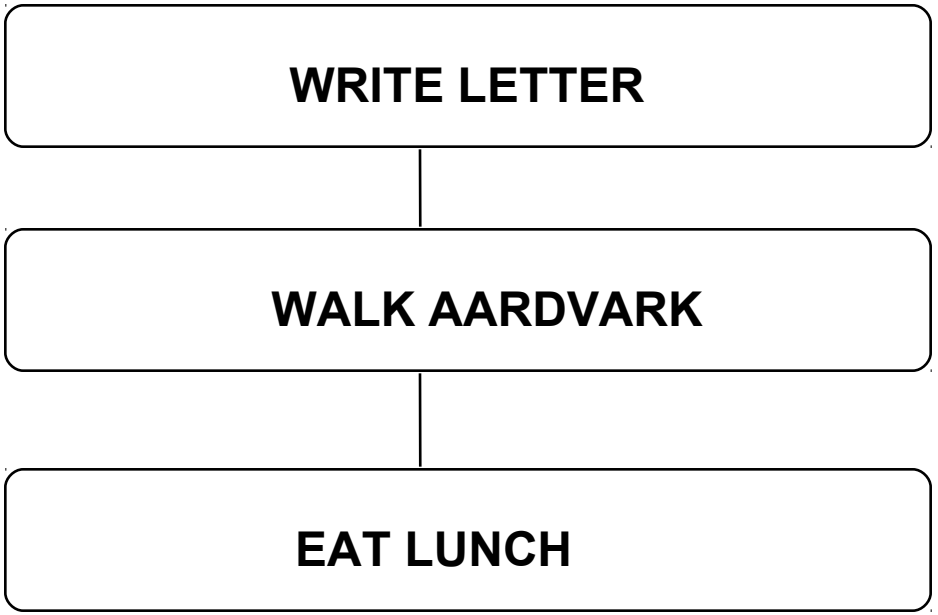
- Superficially sequential & communicational cohesion looks similar
  - However, the primary difference is that a sequential service is an assembly line...
  - ...where control flow must move in a particular order.
  - In a communicationally cohesive service the ordering of the functionality is unimportant.

Communicationally Cohesive Services

- Typically less maintainable than functional or sequentially related activities.
- Why? Temptation to intermingle the code of all activities defined within the service.
  - ...if change is required it can often impact on other activities undertaken by that service.
  - Solution: Often better design to split up a communicationally cohesive service into **n** functionally cohesive ones.

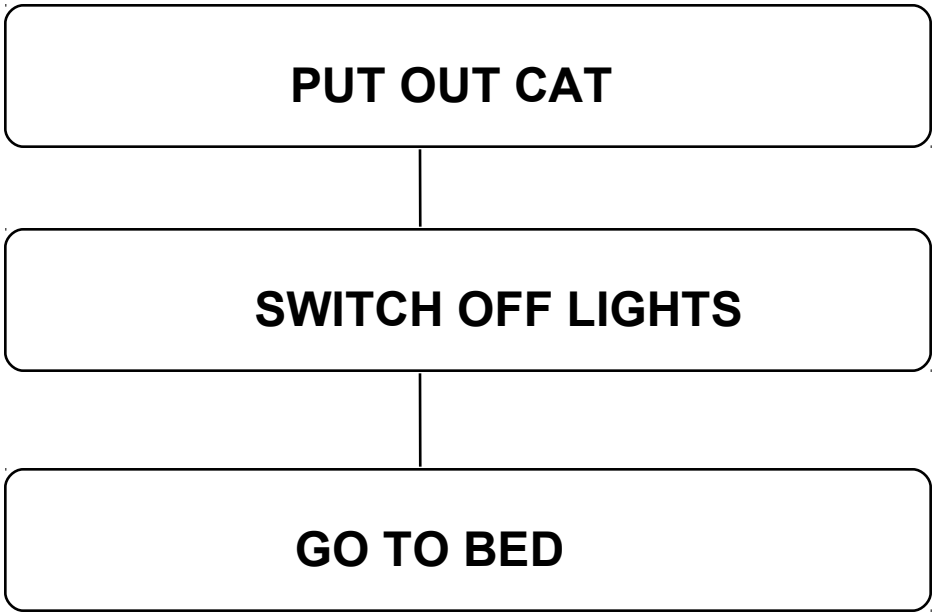
Procedurally Cohesive

- Composed of pieces of functionality that are sequentially organised but...
  - otherwise bear little relationship to each other.



Temporally Cohesive

- Sequentially organises activities that are related by time.



Temporal and Procedural Maintenance

- Temporal & procedurally cohesive services tend to intermingle the code for each of their respective activities...
  - This makes the separation of individual activities difficult, and hence does not reduce the impact of change.
  - Making a change in temporal or procedural ordering means a significant re-structuring of the service implementation.

Logically Cohesive

- Encompasses a set of activities in the same general ‘category’.
  - The activity to be executed is usually chosen by an input parameter.
- E.g.,. Input a number for the required activity
  - Eat Sausage
  - Eat Snail
  - Eat Gorgonzola

Coincidentally Cohesive

- Activities are wholly unrelated. E.g.,
    1. EAT LUNCH
    2. BLAST ALIENS
    3. CLIMB MOUNTAIN
    4. EXPLODE
  - Difficult to understand.
  - Difficult to disentangle - code shared by some or all activities in the function.
  - Consequently difficult and expensive to maintain.

Identifying Cohesion

- Write a sentence that describes what the service does -
  - The structure of the sentence often gives away the level of cohesion the service supports:
  - **FUNCTIONAL COHESION :** A service fulfilling a single function can usually be summed up by a precise verb or verb-object name.
    - E.g. ADD_INTEREST.

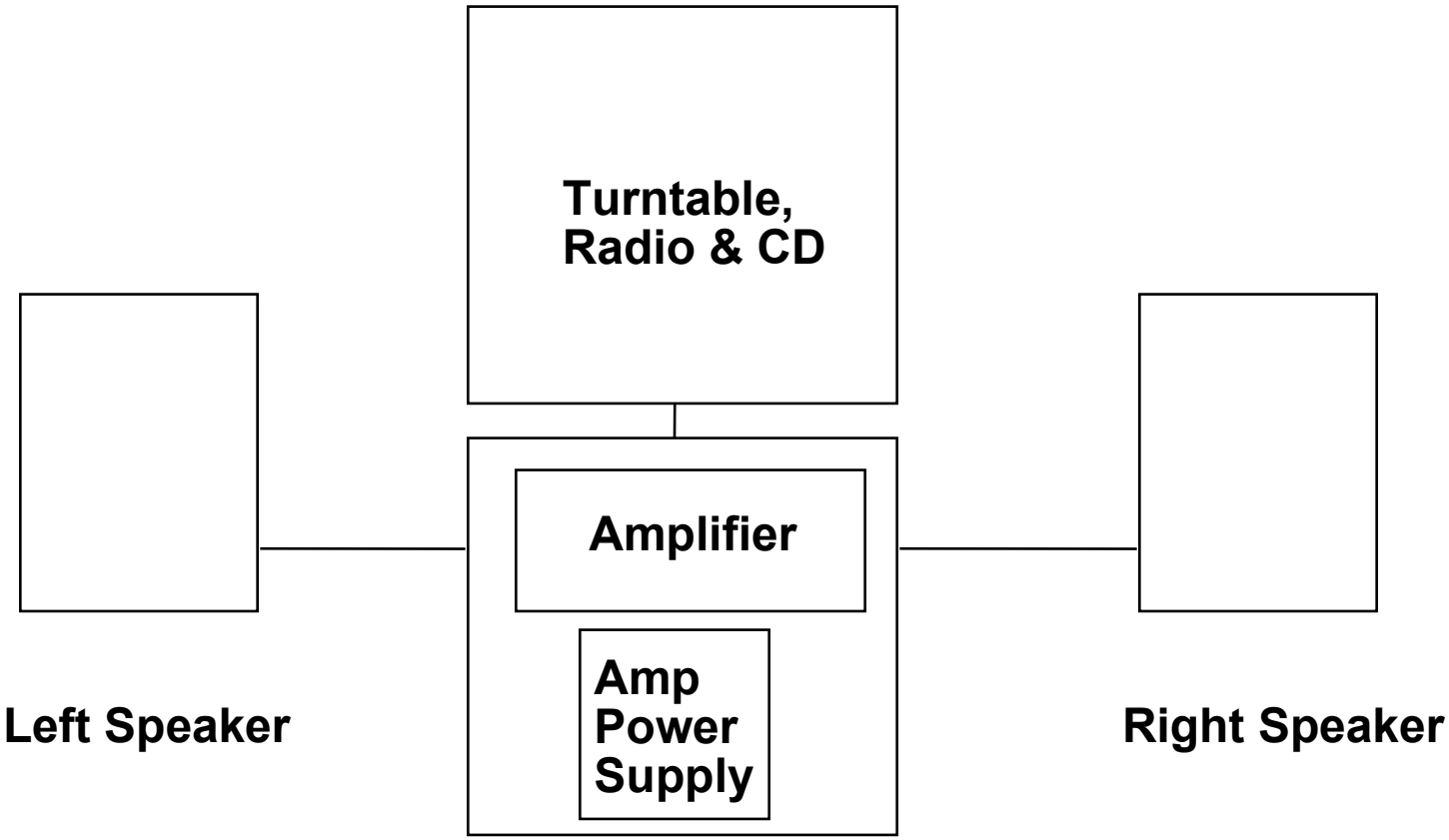
Finding and Classifying Cohesion

- **SEQUENTIAL:** A number of assembly line activities within a service usually demonstrate sequential processing
  - E.g., CONSTRUCT_CAR, UPDATE_AND_VALIDATE_ID
- **COMMUNICATIONAL:** A number of non-sequential activities working on the same data demonstrate this form of cohesion**.**
  - CALCULATE_MONTHLY_AND_YEARLY_VAT_RATE
- **PROCEDURAL:** look for procedural names.
  - E.g., LOOP_ROUTINE , STARTING_THE_DAY

Finding Further Classes of Cohesion

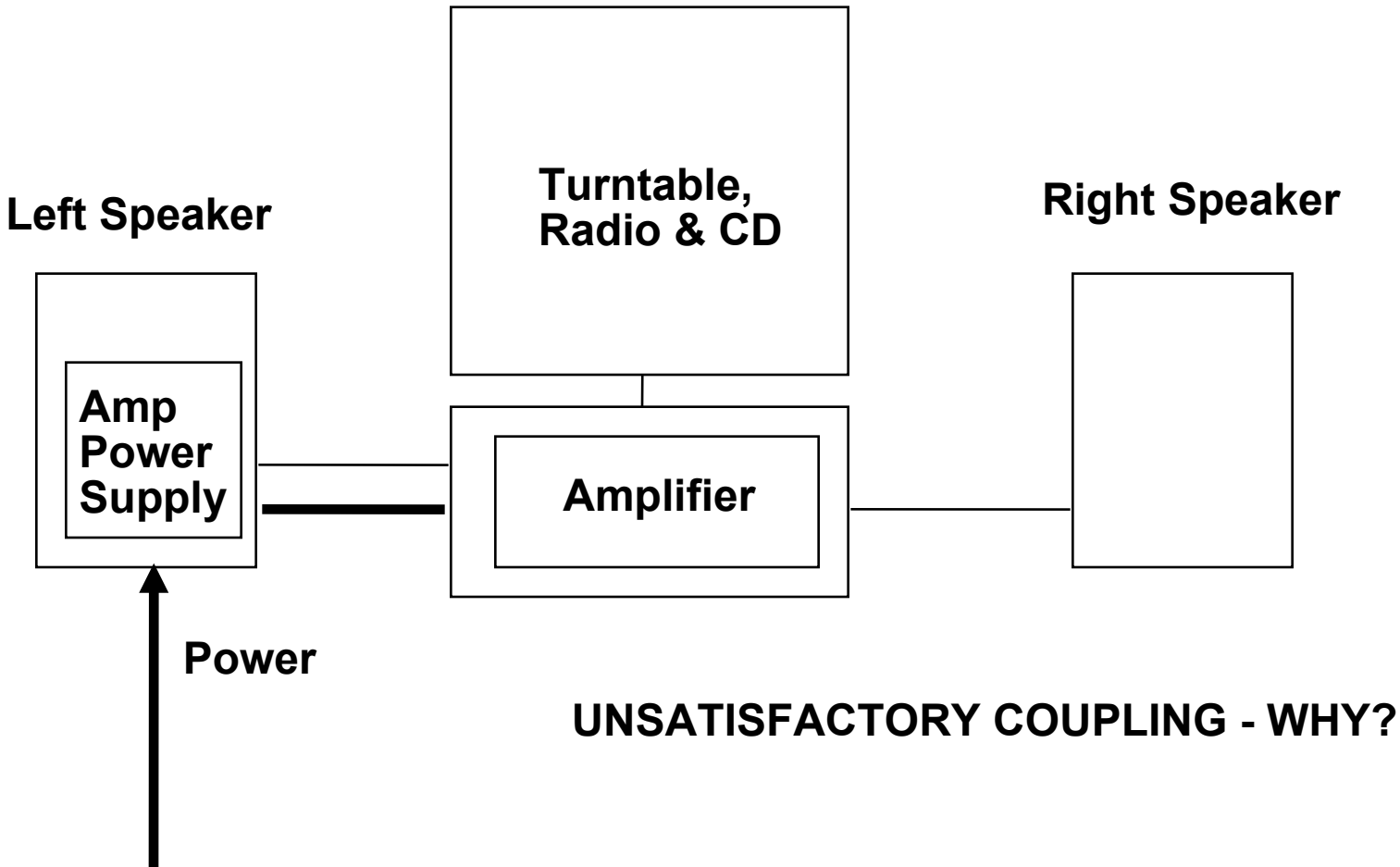
- **TEMPORAL:** Names that have time related semantics are a dead give-away.
  - E.g., START_UP , DO_AT_MIDNIGHT
- **LOGICAL:** Look for general ‘all purpose’ services that do different things with different inputs.
  - E.g. TRAVEL_BY (mode), EAT_SCOFF (food type)
- **COINCIDENTAL:** The describing name doesn't make sense, nor does any clear functional description.

Coupling



LOOSELY COUPLED

Unsatisfactory Coupling



Really bad Coupling

Turntable , Speakers , CD , Amplifier,

Amp power supply , Radio

- SPEAKERS ARE TOO CLOSE TOGETHER - AND I CAN’T SEPARATE THEM.
- THE BOX WON'T FIT ON THE SHELF - ITS TOO BIG..
  - AND WORST OF ALL THE TURNTABLE MOTOR CAUSES THE AMP TO HUM.

Coupling is..

- The degree of interdependence between two objects.
- Minimise coupling, and by doing so make each object as independent as possible.
- Low coupling indicates a well-partitioned system and can be attained by:
  - eliminating unnecessary relationships,
  - reducing the number of unnecessary relationships,
  - easing the ‘tightness’ of necessary relationships.

Why Coupling Works

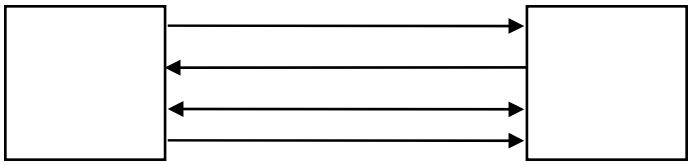
- Fewer connections between objects lessens the chance of a ‘ripple effect’ (a fault in one module causing errors in another).
- Low coupling means that a change in one object is unlikely to require change in another.
  - When maintaining an object don't want to worry (or know) about the implementation of other objects.

Principles for Coupling

- Create:
  - Narrow (as opposed to broad) connections.
  - Direct (as opposed to indirect) connections.
  - Local (as opposed to remote) connections.
  - Obvious (as opposed to obscure) connections.
  - Flexible (as opposed to rigid) connections.

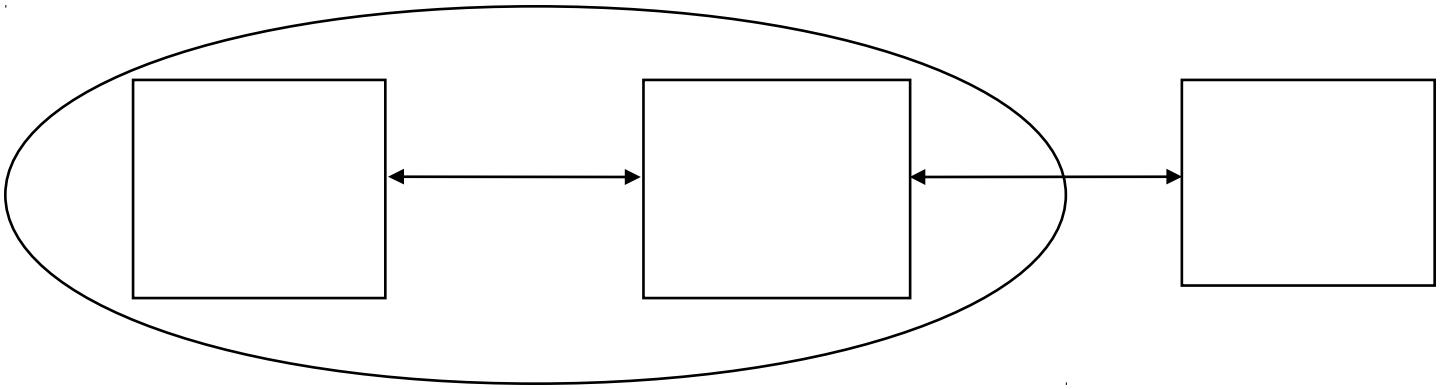
Narrow Connections

- Breadth of interface is essentially the number of connections that link two objects -
  - ideally dialogue between objects should be as ‘narrow’ as possible.
  - The number of messages an object sends to a given object (and the number of parameters) should be kept to a minimum.



Direct Connections

- Coupling between two objects is more understandable (and less complex) if the developer does not have to refer to other objects to understand the original connection.



Local Connections

- In non-OO systems modules should communicate through parameter passing not through the use of global data structures.

Obvious Connections?

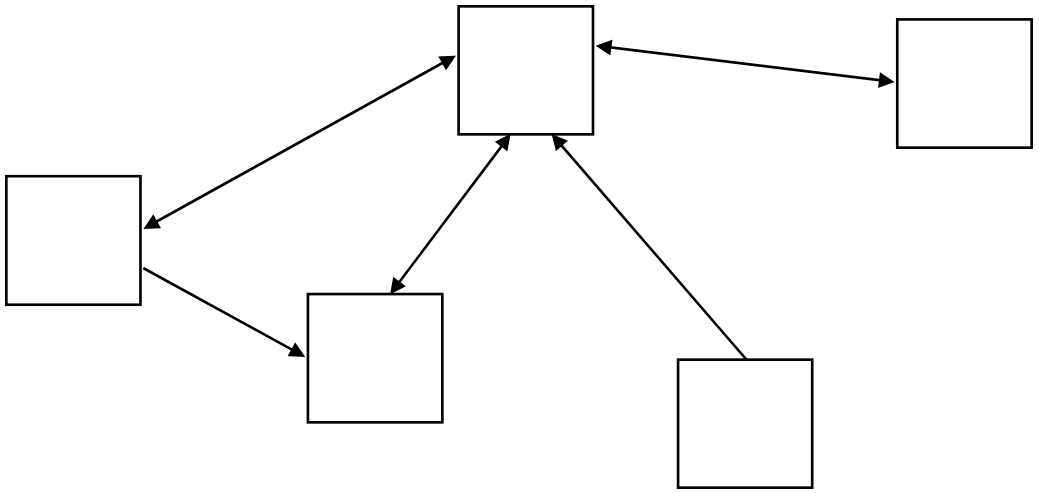
- Make sure that data passed has a name and structure applicable to the service(s) in which it is employed
- Build cohesive data structures which have obvious relevance in the context.

Flexible Connections

- Links between objects often have to be changed as a consequence of maintenance.
  - Costly if links are ‘rigid’.
    - ‘An example of a rigid connection would be one in which a designer had decided that a module should collect the information it needed from a fixed location in memory...
    - ..In that case if you decided to reuse the module in a different part of the system (or port it to another hardware architecture!) you would either have to goof up the previous use of the memory location or write some ugly little wart of code to get around the problem’ [Page Jones].

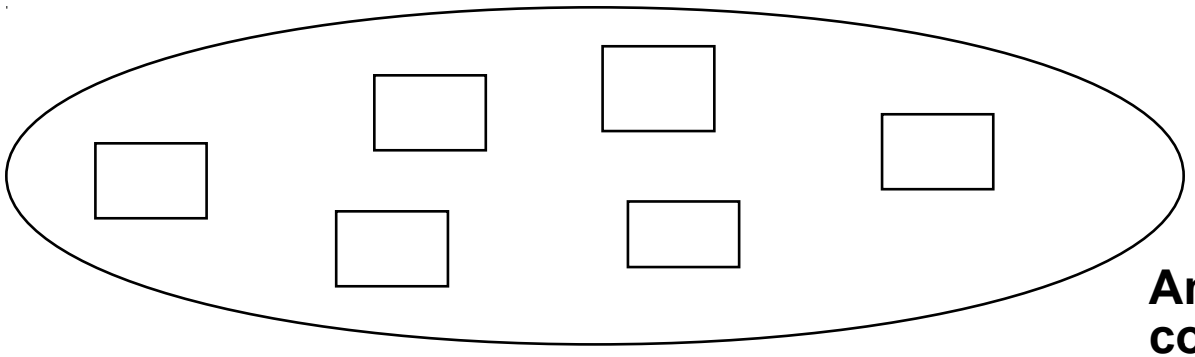
Wired Coupling

- A wired architecture is where all message links between objects are explicitly captured..
  - ..even though objects may be created & destroyed throughout the lifetime of the system.



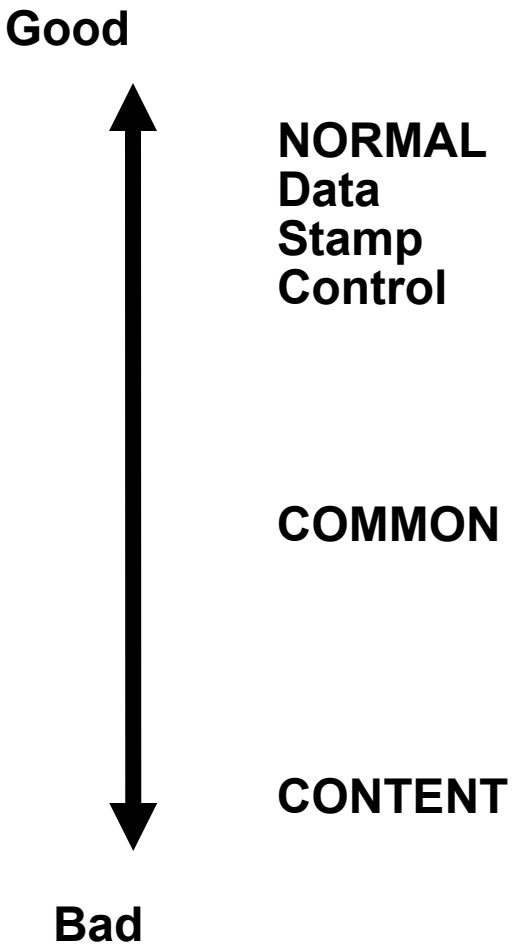
Wireless Coupling

- A wireless architecture is where objects may potentially communicate with other objects by passing flags.
  - In C++ these would typically be class pointers.



Any object may communicate with any other

Types of Coupling



Data Coupling

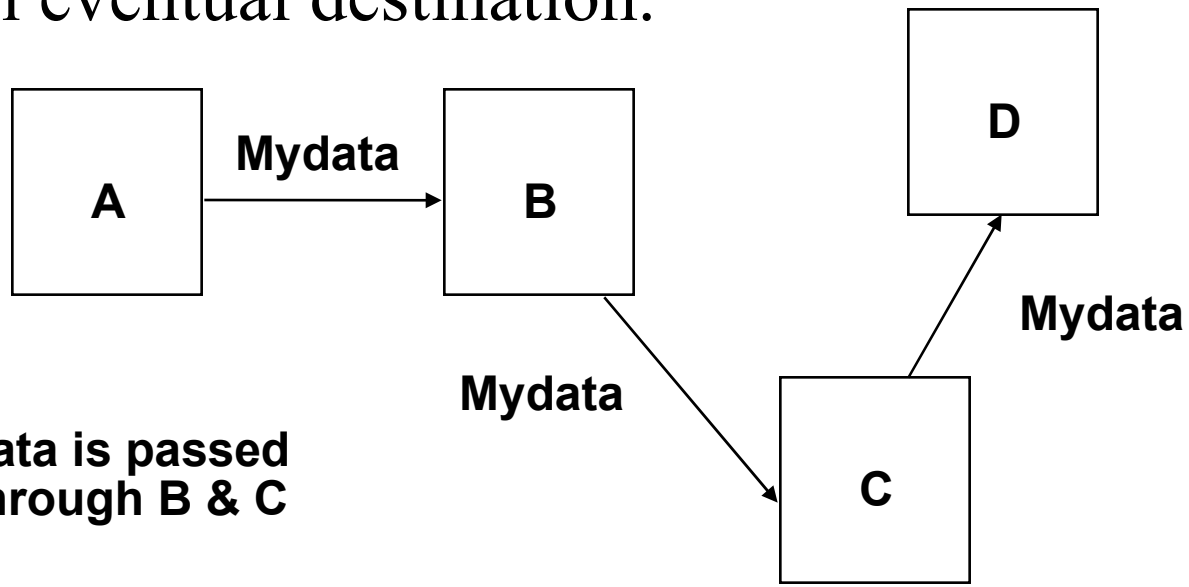
- Most obvious and common form.
  - Two objects pass information through the use of parameters
- Data coupling is local (it passes values directly between two objects).
  - If correctly used the data passed is used directly by the calling and called service.
  - Flexible and maintainable, as long as a service accepts the same parameters it can be changed without adverse consequences.

Guidelines: Narrow please

- Small is beautiful. Keep the number of parameters to a minimum if at all possible.
  - A mob of different parameters crossing an interface is more likely to increase the possibility of errors.
  - Vast amounts of parameters sent to a single service often obscure both its purpose and functionality.
  - In other words build interfaces to be as narrow as possible.

Guidelines: No tramp data

- Avoid the use of ‘tramp data’.
  - Bits of information that are passed from object to object (without being used) until they reach an eventual destination.



Stamp Coupling

- Two objects are stamp coupled if they pass each other composite data forms e.g. records or structs
  - E.g., a Customer record that includes name, number, age, favourite pizza etc.
- Stamp coupling is fine, but the composite data forms add a slight degree of obscurity to the design.

Stamp Warning

- Never pass composite data to an object that needs only one or two fields from that data.
  - Broadens the interface, obscures the design, and (as a consequence) can increase maintenance costs.
  - There is also the possibility that an object service could inadvertently manipulate (and change) values that it doesn't use.
  - In these cases it is better to send the data as individual parameters.

No Bundling please

- If two objects are data coupled, but share a broad interface (i.e., they pass vast amounts of parameters between them) the temptation is to aggregate these into a single data structure.
  - E.g., could aggregate the integers ID_no, age and shoe size into a struct called ‘stuff’.
- This is called bundling.
  - Bundling is undesirable, obscure and un-hygenic.
  - Don't do it just to reduce coupling.

Control Coupling

- One object passes the other a piece of data intended to control its internal logic
  - Usually symptomatic of logical cohesion.
  - E.g., a ‘travel_by’ service that accepts a data flag representing CAR , BOAT, PLANE , etc. from another object.
  - Leads to indirectness and obscurity.
  - Usually a result of poor partitioning -
    - e.g., for one object to ‘control’ another it implicitly knows details of its implementation.

Common Coupling

- Two modules share a common data structure.
  - Not applicable to Coad & Yourdon - but may be relevant to C++.
  - Violates basic principles of encapsulation and modularity.
    - Erroneous updating of global data by a module has ripple effects on all other modules that use it.
    - Such modules are obscure, difficult to maintain, difficult to reuse, any changes to the global data will necessitate change in all modules that use it.

Content (Pathological) Coupling

- Where control flow leaps merrily from object to object through the liberal application of GOTO statements -
  - The sequence of execution may **jump** from one service to the the middle of another (encapsulated by another class-&-object!)
  - Makes a mockery of the entire object-oriented design process..
  - ..and forces an enormous degree of interdependence between objects.

Connascence

- Page-Jones proposes the generalisation of both OO coupling and cohesion into a single measurement called Connascence...
  - ‘I say that two elements of software are Connascent if they are 'born together' in the sense that they somehow share the same destiny’.
  - Note similar move to single measure in RADs.

What is Connascence?

- Page Jones again:
  - ‘I define two software elements A and B to be connascent if there is at least one change that could be made to A that would necessitate a change in B in order to preserve overall correctness’
  - ‘Eliminate any unnecessary connascence and then minimise connascence across encapsulation boundaries by minimising connascence within encapsulation boundaries’.

Type Connascence

```
int x;
    ...other statements...
    x = 7;
```

- ..a change to the type i.e., int to float will impact the later assignment statement.
  - In C++ this would compile but will be incorrectly typed (probably).

Name Connascence

```
int x;
    ...other statements...
    x = 7;
```

- ..a change to the name i.e. x to j will impact all connascent points in the program based on the variable name x.

Other Forms

- Value - two software elements must contain the same values
- Algorithm - two software elements must agree on some common algorithm for their correct execution.
- Semantic - two software elements must have identical semantic as opposed to syntactic structure.

Summary

- Examined forms and problems of Cohesion.
- Examined forms and problems of Coupling
- Described unifying idea of Connascence.