Decomposition of Software Into Components

David Lorge Parnas, P.Eng., Ph.D, Dr.h.c., Dr.h.c., FRSC, FACM, FCAE

SFI Fellow, Professor of Software Engineering
Director of the Software Quality Research Laboratory (SQRL)
Department of Computer Science and Information Systems
Faculty of Informatics and Electronics
University of Limerick

Abstract

Most software products are too large to be completed by a single person in short period. To make the development manageable, the software must be divided into components that can be developed (and later maintained) separately. Each component will be a work assignment for a team or individual.

It is often thought that this decomposition is a management decision, determined primarily by the talent available. This lecture explains that the decomposition is a critical design decision to be made on the basis of simple technical criteria, which will be stated and illustrated. The result is a very unconventional, but easily maintained, design.
Thinking about Large Programs

What is a “large program”?  
• One too large to be written by one person in a few days.

Why are large programs different?  
• There are many tiny details.
• It is difficult, often impossible to keep all of those details in mind all the time.
• When there are several programmers, nobody is familiar with everything in the program.
• If the program is written over a period of more than a few days, people forget the details of what they have done.
• Errors in one part often affect other parts.

How should we respond to these problems?  
• We organise programs into components called modules.
• Production of a module is a piece of work for a programmer or a group of programmers.
• Modules can be subdivided into smaller modules.
• Even individuals organise their work in modules.

Is modularisation an engineering issue?  
• Some treat it as management: assigning people work.
• There are technical issues: issues independent of the people available.
• Dividing a program into modules is the subject of this lecture.
What is a module?

Historically modules were simply a unit of measure.

Manufacturers learned to build parts that measured one such unit as this simplified both their job and that of the customers.

The word now means the parts themselves.

Modules are usually relatively self-contained systems that are combined to make a larger system.

Design is often the assembly of many previously designed modules.

The assembly of a system made of modules is usually best done if the interfaces are simple and well-understood by all involved.
The Constraints on Modules

If the modules are hardware, how you put them together is obvious; there are well-known physical constraints. There is a well-identified time at which modules are assembled to get the larger system.

If the modules are software, there are no obvious constraints. Theoretically software modules can be arbitrarily large, their interfaces arbitrarily complex.

During software development there are several different times at which parts are combined to form a whole.

During software development there are several different ways of putting parts together to get the whole.

The result is that words like “module” and components have become buzzwords in the software industry.
Modules of software--When are Parts Assembled?

A. While writing software
   • **parts**: work assignments for programmer(s)
   • **when**: files containing programs combined before compilation or execution.

B. When “linking” object programs.
   • **parts**: separately assembled (compiled) programs with “relative” addressing
   • **when**: addresses are inserted to provide links before execution.

C. When running a program in limited memory.
   • **parts**: memory loads - data and programs that refer to each other by memory addresses
   • **when**: while the program is running.

The word “module” is used in programming literature for all three of these! The ambiguity in the word leads to confusion. I will talk only about the first meaning.
Constraints on the Three Structures

Write time Constraints:
- Intellectual coherence for programmer
- Ability to understand, verify
- Desire to change each module independently of other module changes.

Assembly time constraints:
- duplicate names
- time to re-assemble

Run time constraints
- size of memory
- frequency of reference to items outside segment
- time to load/unload memory

The three sets of constraints are independent and they have only the word “module” in common. These are three different design concepts.
Myth of Over-Modularisation in TSS/360

TSS/360: a major effort by IBM to build a time sharing system in the 60’s.

- It was very slow.

A well known IBM researcher attributed this to over-modularisation - the modules were *too small* and there were too many.

Previous popular wisdom - modules should be *as small as possible*.

Researcher, “Too many small modules led to memory thrashing”.

Belief in many small modules was based on work assignment interpretation of that word.

Implementation used memory management interpretation.

Two meanings of module were being confused.
University of Limerick

The Effects of Confusing these Meanings

Inefficiency results from forcing coincidence.

Good “write time” modules may not be good “memory load” modules.

Write time modules need not be compiled separately. One may use macro substitution to assemble.

Assuming that a module must call a procedure in order to use another module may make the whole scheme impractical.
The Meaning of Module for the Next Hour

In this lecture, modules are *always* write-time or change-time entities.

I want them to have the following properties:

- They can be designed and changed independently.
- They can be divided into smaller modules.
- They are good candidates for reuse in future software.

When do we stop subdividing modules?

- When they are so small that it is easier to write a new one than to change it.
- When the cost of specifying the interface exceeds the expected benefit of having smaller modules.

The concept of “module” as a work assignment is only a definition.

Designers need criteria for judging a proposed decomposition and guidelines for finding a good decomposition.
The KWIC INDEX Example: Conventional structure

1. Input Module: Reads input into main memory
   • INPUT INTERFACE: Input format, marker conventions
   • OUTPUT INTERFACE: Memory format

2. Circular Shift Module: Prepares rotations of input lines for key words
   • INPUT INTERFACE: Memory format
   • OUTPUT INTERFACE: Memory format, perhaps the same

3. Alphabetising Module: Sorts the rotated lines
   • INPUT INTERFACE: Memory format
   • OUTPUT INTERFACE: Memory format

4. Output Module: Prints the sorted rotated lines
   • INPUT INTERFACE: Memory format
   • OUTPUT INTERFACE: Paper format, conventions, etc.

5. Master Control Module: Invokes the above modules in sequence.
   • INTERFACE: names of the program to be invoked
University of Limerick
The KWIC INDEX Example

Which program design decisions are most likely to change?

1. Input format

2. Memory formats

3. The decision to sort all the output before starting to print results.

4. Output formats
KWIC INDEX Example: An Alternative Structure

Line Holder Module

- A special purpose memory to hold lines of characters

It consists of the following programs:

GET_CHAR (lineno, wordno, charno)

SET_CHAR (lineno, wordno, charno, char)

CHARS (lineno, wordno)

WORDS (lineno)

DELETE_LINE (lineno)

DELETE_WORD (lineno, wordno)
The KWIC INDEX Example: Alternative Structure

Input Module
• reads from input medium;
  • calls line-holder programs to store in memory

Interface program: “INPUT”

Note that this module is simplified because the line holder does some of its work.
The KWIC INDEX Example: Alternative Structure

Circular Shift Module

- Creates a “virtual” list of circular shifts.
- Uses line holder programs to get data from memory.
- It may, may not, create an actual table.

Interface programs:
(1) “CS_SETUP”
(2) “CS_CHAR (lineno, wordno, charno)”
The KWIC INDEX Example: Alternative Structure

Alphabetiser Module
• Does the actual sorting of the circular shifts.
  • May or may not produce a new list.
  • If it doesn’t, it makes a directory.
• May sort in ALPH, ITH or both.

Interface programs:
ALPH
ITH (lineno)
The KWIC INDEX Example: Alternative Structure

Output Module
- Does the actual printing.
- Calls ITH and circular shift programs.

Interface program:
OUTPUT
The KWIC INDEX Example: Alternative Structure

Master Control Module

- Links all the modules together to do the job.
- Is the main program, but very simple.
- Calls INPUT, CS_SETUP, ALPH, and OUTPUT.
The KWIC INDEX Example: Alternative Structure:

What happened?

- Not necessarily getting a really different program.
- Not necessarily different algorithms or data structures.

Different way of cutting up a program

- *Likely* changes are confined to one module.

Design is based on explicit consideration of what needs to be changed.

Simplifies interfaces

- information hiding, abstraction
- descriptions may be less familiar

In the original design, interface were data structures.

In the new design, the data structures are internal, not external.

A change to a data structure affects only one module unless you add new, exported information.
Finding the Information-Hiding modules

Do not:

• think about “use cases”
• think about transactions
• think about the processing steps.

Identify design decisions that are likely to change.
Design a module to hide each such design decision.
Requires experience and judgement. Rare to get it completely right.

For an information hiding design, the requirements document should provide an indication of what is likely to change and what is unlikely to change.

• Programmers may not know this.
• All participants may not agree.

Designing for change in requirements or platform requires knowing what those changes might be. If changeability is a requirement, you must think about changes.
Terminology

The *secrets* of a module: Design decisions that can be changed without affecting any other module.

Line holder
- how lines are represented in memory

Input module
- input format

Circular shift module
- how circular shifts are represented

Alphabetiser
- sorting algorithm
- when the alphabetisation is done

Output
- output format

A shared secret is not a secret.
A module’s secret can be changed without affecting any other module.
Design Errors in the Conventional Design

Flowchart boxes become modules.

There were unnecessary “connections”.
  • All of the modules contained code that is dependent on data structure design decisions.
  • The whole program was written on the assumption of a given input format.
  • The whole program was written on the assumption that printing would be done at the end.
Switching from Module to Module

Steps-in-processing approach
- There are few transfers from module to module.
- The cost of the transfer is not significant.

Information-hiding approach
- There are many separate programs called from other modules.
- There is a high frequency of switching
- The cost of switching can be very significant.

Module access programs need not be subroutines.

The usual space-time tradeoffs apply
This what was overlooked by language designers.
This is part of what the “aspect crowd” have reacted against.
Programming tools are failing us in this regard.
Do not Confuse Run-time with Write-time!

We want to hide information about the design at write-time, not minimise data exchange at run-time.

Reducing the information passed between modules at run-time may actually increase the amount of information exchanged at write time.

You can save run-time information storage and processing by making assumptions at write-time.

This will speed up your program and save memory, but may make it much harder to change.

The module structure is a write-time structure and may disappear with final assembly.

There is a trade-off between flexibility and efficiency but careful design can sometimes reduce the run-time cost of flexibility.

In the years since “my time” efficiency has become less important.
The Error in my KWIC Design

Proposed change:

- Sort words, assign an integer in alphabetical order.
- Sort sequences of integers instead of sequences of words.

Affects many modules: interface reveals that data are sequences of strings.

We could have hidden the fact that the data were strings. -- Why not? Subconscious revealing of secrets is easy.
Design Procedure for Large Programs

A. Identify the secrets, separate as far as practical
   • Decisions that will change together may stay together.
   • Mutually dependant decisions can stay together.
B. Design an information hiding interface for each:
   • Implementations of access programs are based on the secrets.
   • The interface won't change if the secrets do.
C. Specify all the interfaces precisely.
D. Implement independently using only interface information.
   • One module may use access programs from another.
   • Programs from one module cannot use data structures or internal programs from another.
E. Decompose the larger modules - go to A.
The Relation Between Information Hiding and Abstract Data Types

Data Abstraction is a special case of information hiding. Algorithms can be hidden as well. Data Types allow many copies of the hidden structure.

- Each variable has one copy of the hidden data structure.
- The module is the programs.
- The data structure copies are the variables.
The Relation Between Information Hiding and Object-Oriented Approaches.

The modules of the KWIC produced objects.

Modules are the code that produces objects.

Conscious attention to information hiding is what makes O-O work.

Not dependent on message passing as was commonly thought.

Class-inheritance wasn’t there.

You can write terrible programs in O-O languages.
We are designing not one program, but a program family. Decisions shared by all members of a family should be made early. Decisions likely to change should be postponed. Early decisions are harder to change than later ones. Structure decisions are early and hard to change. Interfaces: Should embody decisions less likely to change. Implementation: Should embody decisions most likely to change. The more likely a decision is to change, the more you restrict knowledge of it. Where possible decisions that are likely to be reversed should be made late in the design process. Making a hidden decision early is not as harmful as making an incorrect open decision.
Example Linked Lists

Consider the following data structure designed for easy insertion and deletion:

![Diagram of linked lists]

Consider the following alternative.

![Diagram of alternative linked lists]

The link at each element is the sum of the forward and backward links. Each alternative has advantages and disadvantages. The decision is likely to change.
Example: Linked List, Using Information Hiding

1. Make the job of maintaining the list a module.
2. Provide the following interface programs.
   • Current: returns the value of the current element.
   • Moveforward: current element becomes next on list.
   • Movebackward: current becomes previous on list.
   • Atstart?: returns true if you cannot move backward.
   • Atend?: returns true if you cannot move forward.
   • Insertafter(x): insert a new element with value x in the list after current.
   • Insertbefore(x): insert a new element with value x in the list before current.
   • Remove: Deletes the current element from the list.
   • Alter(x): Changes the value of the current element to x.
3. Write the programs that use this list using only the interface programs.
   • Your programs can manipulate the list without using the link structure directly.
   • Your programs will be easier to understand and change.
   • Your programs are more likely to be correct.
More complex software systems are:

- even harder to understand,
- Hard to maintain,
- Full of small inconsistencies,
- Full of a lot of almost “clones”,
- Full of undesirable coupling between components.
University of Limerick

To Improve Quality

1. We must constantly keep maintenance in mind when designing.

2. We must impose a consistent set of design rules on our designers.

3. We must do something different!
   - What we have been doing doesn’t work.
   - A new conventional design won’t be any better than the old ones.

4. We must design for change
   - The only software that isn’t changed is software that isn’t used.
   - If you don’t consider the future, there may be no future for the product.
How Do You Keep A Secret?

Make the secret uninteresting to the developers of other modules.

- Provide an interface that lets them do what they need to do.
- Provide an interface that does not share the secret
- Provide an interface that will not have to change if the secret changes.
- Provide a precise interface specification.
- Provide a complete interface specification.
- Provide an accurate interface specification.

If other programmers can learn everything that they need to know from the specification, they won’t care about the secret and won’t ask to know it.

One must inspect code to make sure that correctness does not depend on the secrets of other modules.
What is Different About More Complex Systems?

There are many implementation decisions.
There are many details.
There are many designers.

This leads to many issues:
• How can we keep the project under intellectual control?
• How can we maintain conceptual integrity?
• How do we deal with unstructured lists of modules?
• How can we tell when we have them all?
• How does everyone remember the names?
• How do we avoid (near) duplication or cloning of code?
Controlling Complexity If There Are Many Modules?

Group modules into classes characterised by some simple property.
  • Keep each class small and understandable.
  • Make sure the classifications are mutually exclusive.
  • Make sure the classifications are complete.
    • (The classes should be partitions!)

This will:
  • Put some structure in the list of modules.
  • Help to check for completeness.
  • Leads to more helpful naming conventions.
  • Makes duplications less likely.
  • Make a specific module easier to find.
The Module Structure Must be a Hierarchy with Several Levels

We cannot check long lists for completeness.
We cannot check long lists for lack of overlap.
Modules should be small and have only one secret.

- There will be many modules.
- We may need many levels.
Some Possible Classification Criteria for Modules?

A. By similarity of interface
   • too vague
B. By ultimate purpose
   • can be ambiguous
C. By type of “function” or service provided
   • too vague
D. Similar programming problems
   • implementation dependent
E. By nature of the secret
   • limited to information hiding designs
   • Must be unambiguous
   • Requirements will be used to disambiguate when needed.

The only one that works is (E).
The others lead to shared secrets and multi-secret modules.
What is a “Module Guide”? 

It is a document produced by designers for maintainers. It describes the module structure in a way that makes it easy for maintainers to find the module that they are looking for to fix a problem or make a change. It guides you level-by-level through the decomposition hierarchy. At each level it classifies the “secrets”. It is tempting to describe the role a module plays or its function. 

Services and roles cannot be the secret of a module.
What does a Module Guide Look Like?

We will discuss an example.
The example is neither toy nor huge.
The example is not likely to be like your problem.
The upper levels are generic for complete systems.
The lower levels are specific to application areas.
The example is the Onboard Flight Program (OFP) for an old aircraft (A-7E).
The SCR A-7E Module Structure

Top-level decomposition

1. Hardware-hiding module
2. Behaviour-hiding module
3. Software decision module

If the secret is in the software requirements document, it must be (1) or (2).
If it is not a requirement, it must be (3).

Today, hardware includes software implemented “virtual machines”.
This decomposition is based on a clear description of the requirements.
The secrets of module 1 should be in hardware/support software manuals.
All secrets of module 2 must be visible outside the system.
The secrets of module 3 are the truly secret secrets.
However, all modules keep secrets from each other.
The SCR A-7E Module Structure
Second-level decomposition

1. Hardware-hiding module decomposition
   1.1 Extended computer module
   1.2 Device interface module

If something applies to more than one device, consider it part of the computer.

- The “devices” can include other systems.
- The “computer” includes support systems that may already hide the hardware computer.
- Some device capabilities/limitations cannot be hidden but can be parameterised.
- These module must not “know” what the devices are used for - that’s behaviour hiding. (One secret, no shared secrets).
The SCR A-7E Module Structure
Second-level decomposition

2. Behaviour-hiding module decomposition
   2.1 Function driver module
   2.2 Shared services module

- There will be one function driver for each controlled variable.
- Some judgement is required in identifying variables.
- If some behaviour should be consistent in two modules, move it to a shared module. Avoid duplication and shared secrets!
- Watch out for coincidences.
- Secrets cannot be added at a new level. The upper level descriptions must remain valid. Revise upper levels if they were wrong!
3. Software decision module decomposition
   3.1 Application data type module (objects)
   3.2 Physical model module (encapsulate models, allow improvements)
   3.3 Data banker module (secret is update policy)
   3.4 System generation module (reusable tools)
   3.5 Software utility module (more general tools).
1. Extended computer module decomposition
   1.1 Data type module
   1.2 Data structure module
   1.3 Input/output module
   1.4 Computer state module
   1.5 Parallelism control module
   1.6 Sequence control module
   1.7 Diagnostics module (restricted)
   1.8 Virtual memory module (hidden)
   1.9 Interrupt handler module (hidden)

Note: Independent change is unlikely but possible simplification results from separation.

What we are hiding is hardware, not the algorithms for doing these things. Those are software decision modules that these modules may need to use.
Third-level Decomposition

2. Device interface module decomposition

   2.1 Air data computer
   2.2 Angle of attack sensor
   2.3 Audible signal device
   2.4 Computer fail device
   2.5 Doppler radar set
   2.6 Flight information displays
   2.7 Forward looking radar
   2.8 Head-up display (HUD)
   2.9 Inertial measurement set (IMS/IMU)
   2.10 Panel
   2.18 Waypoint information system
   2.19 Weapon characteristics
   2.20 Weapon release system
   2.21 Weight on gear

Note: Almost corresponds to hardware structure, but not quite. Exceptions are closely linked devices.

- Secrets are hardware characteristics, not purpose of device or algorithms.
- Each module creates a virtual device.
Third-level Decomposition

3. Function driver module decomposition
   3.1 Air Data Computer functions
   3.2 Audible Signal functions
   3.3 Computer Fail Signal functions
   3.4 Doppler Radar functions
   3.5 Flight Information Display functions
   3.6 Forward Looking Radar functions
   3.7 Head-up Display (HUD) functions
   3.8 Inertial Measurement Set functions
   3.9 Panel functions
   3.10 Projected Map Display Set functions
   3.11 Ships Inertial Navigation System functions
   3.12 Visual Indicator functions
   3.13 Weapon release functions
   3.14 Ground test functions

- Input-only devices are missing!
- These modules are device independent, they have no hardware dependence.
Third-level Decomposition

4. Shared services module decomposition
   4.1 Mode determination module
   4.2 Stage director module
   4.3 Shared subroutine module
   4.4 System value module
   4.5 Panel I/O support module
   4.6 Diagnostic I/O support module
   4.7 Event tailoring module

These are all behaviour hiding modules. They hide no software decisions or hardware characteristics. They hide information that appeared in the requirements document.
Third-level Decomposition

5. Application data type module decomposition

Examples:
- Angles (several versions)
- Distances
- Temperatures
- Local data types for device modules
- STE (state transition event) variables

All of the above are objects.
- These are software decision modules; the secret is the representation of the data.
Third-level Decomposition

6. Physical model module decomposition
   6.1 Earth model module
   6.2 Aircraft motion module
   6.3 Spatial relations module
   6.4 Human factors module
   6.5 Weapon behaviour module
   6.6 Target behaviour module
   6.7 Filter behaviour module

• The secrets are software design decisions.
• What is hidden is not the shape of the earth or the behaviour of the weapon, but how we model it.
Third-level Decomposition

7. Data banker module
   • One for each real-time data item
   • Value always up-to-date
   • Secret: When to compute up-to-date value

Note that these are neither behavioural requirements nor hardware characteristics. The secrets are decisions made by software developers.
Documentation

- The module structure must be documented in a module guide.
- Each module must have a precise and complete interface description.
- Each implementation of a module requires a module internal design document.
- Documents must be kept “alive”.
- Engineers will take advantage of mathematics in the documentation.

It won’t work without good documentation.
You won’t be able to write the document unless the design is good.
The benefit is not only for maintainers; The design will be better if you produce the documents first.
Conclusions

In complex systems, Information Hiding can be carried out consistently.
Classify into small, obviously complete, non-overlapping lists.
Show secrets not interfaces or roles.
Requirements Document essential for disambiguation.
Devices Interface modules hide IN and OUT.
Function Driver modules hide NAT and REQ.

Each of the resulting modules creates one or more objects (variables of a newly defined, abstractly specified, data type).

Documentation essential!

- A module guide
- Module Interface Documents
- Module Design Documents (per design)