Software Design Document

Team USA
Software Engineering
Sam Houston State University

October 22, 2015

Contents

1 Design #1: Top Down Iterative Refinement

1.1 General Considerations

1.2 Metrics

1.3 Analysis

1.4 Conclusion

2 Design #2: Dynamic Data Flow Analysis

2.1 General Considerations

Dynamic data flow analysis is a design methodology used to demonstrate the flow of data through a software system. A good software design, when analyzed with the dynamic data flow method will show a well defined input, or afferent, branch where data is collected and acted upon by related input processing functions (such as opening a file or reading the file contents to an array for use by another module). Data processed by the afferent branch should then be passed to a well defined central transform branch where the core operations of the software are performed on the data, and output data is generated and returned. Finally, this output data is sent a well defined output, or efferent, branch, where it is processed into its final output format, such as a report.

Two data flow diagrams are used when plotting the path of data from input to output in a software system. The first is a flow diagram, which shows data coming into the system, how it moves through the system's various modules, and eventually is output in a useful way. The second diagram is a hierarchical diagram that shows the modules that make up the system, and how they are related. This diagram shows the flow of data, but also implies the control structure of each module as it relates to the others and to the system as a whole. Data couples and control couples are indicated here as well. Further, because this type of analysis focuses on the flow of data, with an implied control structure and without any concern for the contents of the actual function bubbles present in the diagram, it is not able to determine or show the cohesiveness of any particular module. This is detrimental because it potentially allows modules that are merely coincidentally cohesive to exist until the implementation phase, when the problems associated with that form of cohesion will likely begin to appear.

2.2 Analysis Metrics

2.2.1 Coupling

Dynamic data flow analysis focuses on two types of coupling: data and control.

Data Coupling Data coupling occurs when one module depends on the output of another, and the data they exchange is composed of discreet, elementary units, providing exactly what each module needs and nothing more or less. This is the lowest form of coupling with the least dependency between coupled modules.

Control Coupling Control coupling occurs when one module's output is used to influence or determine the execution logic of a subsequent module - for example, a boolean value or control flag is passed from one module into another that performs a different action depending on the value it receives. This type of coupling is detrimental to the software system because it implies that each of the coupled modules must

know something about what is contained in the other, meaning one or both may not be a true black box, which indicates that one or both may not be separately implementable or maintainable.

2.2.2 Miller's Law

Miller's law tells us that, on average, the largest number of tasks that can be simultaneously held in memory is 7 ± 2. Thus, we should work for a design wherein each module complies with this law and its value. Using dynamic data flow analysis, we can see how the data flows into and out of each module, and with the hierarchical model, how control is implied to flow between modules as well. A good design will show, on average, each module having 4 to 5 tasks, and not more than 7 ± 2.

2.2.3 Graciunas's Law

Graciunas's law concerns itself with the number of control relations between modules, and predicts how this impacts the complexity of the software system. For a given set of modules, R, we have that $R = M(M^{2-1} + M - 1)$, where M is the number of subordinate modules. Because dynamic data flow analysis shows the flow of data and the implied flow of control throughout the system, we can use this law and its equation to minimize the number of inter-module relations to the ideal minimum, represented by the equation

$$I = \frac{N(N-1)}{2}$$

where I is the number of interactions between modules and N is the number of modules in the system.

2.2.4 Factoring

Factoring of a system occurs when the system is divided into an upper level of control modules, controlling a lower level of operations modules. Systems that are highly factored have few upper level modules and more lower level modules, and take on a hierarchical form.

2.2.5 Scope of Control

2.2.6 Black Boxes

A module is defined as a true black box whenever the module can be fully utilized with no regard for its construction, and when it performs the same action or function for every invocation, without regard to its inputs or outputs. Dynamic data flow can show the existence of black boxes in a design by showing how the inputs to a module relate to its outputs.

2.2.7 Fan-In/Out

Fan-in and fan-out occur when a module is a target for multiple, higher level modules, or when a module outputs to multiple, lower level modules. Fan-in is desirable because for multiple modules needing the same function, that function can be coded, debugged, and maintained once, no matter how many modules it services. However, to insure reduced complexity and cost, the module that is the target of fan-in should be a terminal module, and should be a true black box - it should perform the exact same function for every module it services. Fan-out is less desirable, because it implies a linkage between a servicing module and the serviced modules. A change in the servicing function may require changes in the modules being serviced. Further, modules with are both targets of fan-in *and* fan-out are the least desirable, as there is an implied linkage between the servicing module and the serviced modules, and because the target module becomes subject to Graciunas's Law, causing complexity to rapidly increase.

2.3 Analysis

The software system in question is that of a point-and-click, two dimensional game. We will draw conclusions from our analysis using both the data flow diagram and the hierarchical diagram from the previous dynamic data flow analysis of the software system.

2.3.1 Coupling

The design exhibits high levels of data coupling between modules, and a low level of control coupling between modules. Data coupling, while potentially undesirable, is the least detrimental form of coupling, and is likely present in every software design to some degree. In general, our design is data coupled at interfaces between higher level modules and lower level modules, and the data is being passed down the hierarchy for efferent modules and up the hierarchy for afferent modules, as expected. Control coupling occurs in the design at interfaces between controller-type modules and the modules they control. Because the system is a game, it relies on certain modules to control the overall execution of the other, subordinate data processing modules. These control modules then are necessarily coupled to their subordinate processing modules. While this is undesirable from a design standpoint, it is unavoidable.

2.3.2 Miller's Law

The design does not exhibit any modules that exceed 7 ± 2 interactions. Of concern is the `Engine State` module, which has 7 links to other modules in the system. Any further requirements on this module will therefore likely result in rapid increases in implementation, debugging, and maintenance & modification difficulties. However, we know that it is desirable to maintain an average of 3 to 4 interactions between modules throughout the system as a whole, and we can perform basic math to determine where our system stands, based on the data flow diagrams we have obtained through our analysis. Thus, we can derive a table showing each module and the number of interactions it has with other modules:

Get Mouse State	3
Get Save Slot	3
Parse Save File	2
Load Level	2
Engine	2
State Manager	2
Engine State	7
Update Player	6
Compare Player & Actor Positions	2
Process Hover Event	2
Process Click Event	2
Update Actor	2
Respond to Event	4
Display Actors	2
Play Audio	2
Render Player	2
Total:	45

So we can calulate the A, the average number of interactions between modules with regard to Miller's Law by dividing 45 by the number of modules N, where $N = 16$:

$$A = 45 \div N = 45 \div 16 = 2.8125$$

Therefore, while two of the design's modules approach the limits of Miller's Law, the overall average number of interactions between modules with regard to Miller's law is well within the expected range of values, at 2.8125.

2.3.3 Graciunas's Law

2.3.4 Factoring

2.3.5 Scope of Control

2.3.6 Black Boxes

2.3.7 Fan In/Out

2.4 Conclusion

3 Design #1: Top Down Iterative Refinement

3.1 General Considerations

3.2 Metrics

3.3 Analysis

3.4 Conclusion

4 Design #1: Top Down Iterative Refinement

4.1 General Considerations

4.2 Metrics

4.3 Analysis

4.4 Conclusion

5 Appendix A: Diagrams & Figures