Review of Parnas’ Criteria for Decomposing Systems into Modules

Zheng Wang, Yuan Zhang
Michigan State University
04/19/2002

What We Will Discuss ……

• Principles - Parnas’ paper “On the criteria to be used in decomposing system into modules” in 1972
• Examples - Some related works using or extending the decomposing criteria with and without citation.
• Conclusion - The affect of Parnas’ criteria on modern software engineering.

Criteria for Decomposing

• Generally, information hiding and changeability enhancing
• Specifically, 5 criteria mentioned by Parnas
  – Define major data structure in a single module
  – Keep instructions calling a routine and the routine itself in the same module.
  – Hide control blocks for runtime environment in a single module instead of interfaces
  – Hide accessorial data in a single module
  – Try to keep processing sequence of a certain item within a module
Comparison of conventional and suggested decompositions

Conventional Decomposition
• The major data structure changes cause the whole system updated.
• The interface between modules are complex. Developers need more time to work together.
• Only works for systems with less than 10,000 instructions.

Parnas’ Decomposition
• Each module knows little about others.
• Managerial - Developers can work independently.
• Product Flexibility and Extensibility - Part of the system can be extended and modified without affecting the other parts.
• Each module in the system can be studied separately.

Example 1 – Managing Abstraction-Induced Complexity (1993)

Goal: Design modules to achieve good performance without sacrificing flexibility and extensibility.

Five implementation modules:
1. Fixed
2. Adaptive
3. Adjustable
4. Open
5. Incomplete

Advantages for Each Model
• Fixed: Single interface for all clients; Simple and good performance if matches the client’s demand.
• Adaptive: Implementation is given based on client’s demand. System can be tuned by itself.
• Adjustable: Clients choose implementation by passing usage hints. Good performance when cases can be parameterized.
• Open: Clients optimize implementation By adding code to the module. Very flexible.
• Incomplete: Client implements missing part and provides tuning. Ultimate flexible.
Disadvantage for Each Model

• Fixed: Interface can only be used in a special field. No other usage beyond the functions provided by the impl.
• Adaptive: Performance depends on how much info. is abstracted.
• Adjustable: Involve clients adjustment from time to time; Fixed set of choices; hard to identify particular circumstances.
• Open: The injected code has to specify what is to be done and how; inject side effects; internal detail is exposed.
• Incomplete: clients have to provide tuning; harder to develop new implementation based on the low-level service.

Example 2 – Why it is hard to build system out of existing parts? (1995)

Goal: Build an environment generator by reusing four software infrastructure to show architecture mismatch affecting composition and causes.

Encountered Issues:
• Code is excessive large
• Slow and hard to maintain
• Needs modification from outside before working
• Recompilation takes time and error-prone

Analyze the Problems

The root reason: Component runs on some assumptions about the structure of the environment that are in conflict with each other when they are combined together.
• Assumption about the nature of the components, violates criteria 1: Major data structures in different modules;
• Assumption of which component control main thread, violates 5: More than one components control sequence
• Assumption about the nature of the connectors, violates 3: Cannot Hide the control blocks for runtime environment in a single module instead of interfaces
• Assumption about the steps in the construction process, violates 2: Each module is instantiated by its own order
Example 3 – Approach to Design Reusable Real-Time Software (1996)

Goal:
A systematic and analytical method to partition a comprehensive algorithm into software components with small-effect operations hence to increase the reusability and reduce the impact of changing.

Two rationales:
• Large-effect components are less likely to be reused than the small-effect ones since more modification involved.
• The impact of changing should locate in the minimum components, if not the same one

5 steps to decompose the large-scale solution to small-scale components with respect of expected changes
1. Identify the algorithmic characteristics
2. Decompose large-effect operations
3. Determine the expected changes
4. Group operations affected by the same changes into the same component
5. Add necessary control components

Conclusion: It is easier to reuse a componentized solution by localizing the parts affected by a particular change within one module.

Example 4 – Architecture based approach to build self-adaptive software (1999)

Goal:
Illustrate decomposition design roles in planning, interoperating, monitoring, evaluating adaptation to retain full flexibility and extensibility throughout the lifecycle.

Design features:
• Software agents to carry out the adaptation
• Explicit representations of components, connections and environment where the adaptation is deployed
• Message/event services to connect adaptation managers to adaptive systems.
Principle and Implementation

Principle:
Let the adaptive software architecture view system as networks of components connected together by connectors.

Implementation:
• Embedded observers in the application to notify the exceptional events.
• An event pattern is used to model application behavior abstractly.
• Event pattern acts as an expectation agent to correspond to the embedded observer.

Example 5 – Software Restructuring by Enforcing Localization and Information Hiding (1992)

• Goal:
Implement a restructuring algorithm to help user understand, maintain and reuse existing system written by using imperative languages.
• Two key principles:
  – Information hiding
  – Localization – the process of collecting logically related computational resources in one physical module.

5 steps of the algorithm

1. transform source code into an abstract syntax tree, identifying primitive semantic information
2. group global variables and functions into package
3. group locally called functions and calling function to the same group
4. based on the relations among functions, organize groups into hierarchical package structure
5. iteratively executing step 2 to step 4 to group the variables and functions into appropriate “packages”
Our Conclusions

Parnas’ information hiding and changeability enhancing, as the fundamental decomposing criteria, apply to everything from the largest architectural concerns to the smallest coding decision. This philosophy guides the modern development of software component, reusable software, adaptive software, and the design of software hierarchical structure.