The Chunking Pattern

David M. Weiss
Iowa State University
Ames, IA, USA
weiss@iastate.edu

Audris Mockus
Avaya Labs Research
Basking Ridge, NJ, USA
audris@avaya,com

***Abstract*—Chunks are sets of code that have the property that a change that touches a chunk touches only that chunk. The pattern described in this paper defines chunks, indicates their usefulness, and provides an algorithm for calculating them.**

***Index Terms*—chunk, modification request, mining software repositories, modularization**

I. Pattern Name: Chunking

This pattern defines chunks, discusses their utility and shows how to identify them.

II. Problem

How to identify uncoupled pieces of software (chunks) that each represent a module on which an individual or a small team can work independently. A key principle of software design is information hiding, which provides a method for organizing software into independently changeable modules. The modular structure of a well-designed system should be stable over the set of anticipated changes during the lifetime of the system. Accordingly, it would be of great interest to be able to identify the modules in the source code that form the system and observe how well they maintain their independence over the lifetime of the system, as changes are applied to the code.

We define a chunk to be a set of code that has the property that a change that touches that set of code touches only that set of code. A module designed to be independently changeable should manifest itself as a chunk over the lifetime of the system. If a chunk remains stable over time, we may think of it as the representation of an information hiding module. Note that chunks that grow over time, or perhaps split into separate chunks over time, that is, that are unstable over time, are likely an indication that the code may need to be remodularized. A system in which the chunks remain stable may be considered to have a model modular structure that is worth studying and reusing. Note that whether the chunks are identified manually or by an algorithm, it is important to quantify how isolated and how stable over time the chunk is.

We propose to measure the isolation of a chunk by counting software changes that cross the chunk boundary, i.e., changes that modify code both inside and outside the chunk. Changes in the code are accomplished by commits to the repository in which the code is kept. A commit that is done for a single task would not cross a boundary of a module that is truly independent of the rest of the system. Thus, unlike call graphs, or data flow graphs, such commits would be the most direct empirical measure of interdependence among parts of the system.

Several difficulties hinder the identification of chunks. First, one must have available information about the set of changes that have been made to the software over time, minimally including which lines of code or files were touched by which change, the reason for making the change, and when the change was made.

Second, changes in system functionality are sometimes distributed over a set of modules, with the changes to each independent of the changes to others, but all contributing to the addition of new functionality. Bug fixes, however, are much more likely to be localized, and conducting an analysis of where bug fixes have been made may help reveal the modular structure more easily.

Third, it is unlikely that chunks will be perfect, i.e., demanding that all changes that touch the chunk be confined only to that chunk is likely to lead to the conclusion that the system is composed of just one or a very few chunks.

This pattern shows how to conduct an analysis of the set of changes made to a software system over time so as to be able to identify chunks. We may think of the chunks as empirical information hiding modules.

III. Solution

1. Collect the set of modification requests (MRs) that each represent a change made to the software. Validate the MRs in two ways:
   (a) that each MR represents a change made for a single purpose and
   (b) that each MR minimally and accurately includes
   - what files the change touched,
   - when the change was made,
   - a description of the change that allows it to be classified as adding new functionality, fixing a bug, or enhancing system structure or properties that do not add functionality but may add to maintainability, performance, or other non-functional system attributes.

MRs are generally used in conjunction with version control systems (VCSs), such as SVN or GIT, so each may be mapped to a (set of) commit(s) to the VCS. Modern VCS's also allow grouping changes into a single commit, and the purpose of the commit may be inferrable from the commit message.

2. Each MR may be related to multiple files. As such, an

MR induced relationship is a hypergraph with each edge connecting multiple nodes (files). It is unlike, for example, a call flow-induced graph, where each function call is a regular edge that connects only two nodes (files). As a consequence, there exists no distance function defined between two nodes that reflects the number of MRs crossing the chunk boundary. Existing techniques, e.g., clustering, require such a function. We, therefore need another algorithm to identify chunks[1]:

a. Randomly select a set of files to be the current candidate chunk, and identify the MRs (or commits) associated with those files, i.e., the MRs that resulted in a modification to those files. Also, decide upon the smallest and the largest size of the chunk you would like to consider, for example by assigning maintenance effort to each file. The simplest approach would be to use lines of code or past changes as a proxy of maintenance effort. Then iterate:

   1) Randomly pick a file F.

   2) Randomly choose step 2a or 2b:

      a. If F is not in the set of the candidate chunk, add it; if inside, remove it from that set, unless the effort assigned to the candidate either exceeds or is below the bounds set above. If these bounds are breached, revert to the earlier state and go to step 1)

      b. If F is outside the chunk, randomly pick a file G inside, if F is inside, then randomly pick a file G outside, then switch F with G. In other words, if F is inside, remove it and add G to the candidate set. As in step 2a, verify that effort constraints are satisfied, if not abort the change and go to step 1).

   3. Calculate the coupling of the candidate to the rest of the system by calculating the number of MRs (or commits) that cross the candidate's boundary. If the coupling measure improves upon the previous best record, save the value of the measure and the set of files in the chunk as the current solution.

   4. If the coupling measure improves then proceed to step 1). If not, then revert to the candidate set before step 2) with some probability greater than zero (and lower than 1).

   5.Stop iterations after a large number of steps.

Enforcing bounds to chunk size makes sure that trivial solutions, such as the entire system or an individual file, are not provided as optimal. This is particularly useful in situations where one is seeking to distribute the work of sustaining a system across a number of different sites, some of which may be widely separated. See [1] for an example.

[1] This algorithm is a variation of the algorithm described in Mockus and Weiss [1].

IV. Consequence

The solution identifies independently changeable pieces of software, and may provide the anticipated amount of work needed to maintain each. It may provide pointers to unstable chunks if the algorithm is applied to increasing time intervals over the lifetime of the system, i.e., tracing the evolution of the chunks over time to see if they remain stable or not.

V. Discussion

The best way to identify chunks may depend on the particular development practices of a project. It is advisable to use only MRs or commits that represent a single task and are changing tightly interdependent parts of the system. For example, fixes to a single bug or an implementation of a single feature may be considered a single task. For some candidate chunks the number of MRs touching the candidate may be so small that they are not worth considering as chunks. This becomes particularly relevant as the system ages; chunk identification is most useful for parts of the system where there is considerable churn, since it is in precisely those parts of the system where it must be easiest to make changes in order to sustain the system, and, for commercial systems, to produce revenue from making changes.

Not all MRs should be considered in calculating candidate chunks. MRs that represent global changes, such as changes to header files, may touch so many files that it is not useful to include them in the chunk calculations. MRs that represent a trivial effort to make a change may also not be useful as they contribute little to the change effort. It is important to ascertain that project practices are ether compatible with MRs representing individual tasks, or, if not, find a way to separate a subset of MRs that do represent individual tasks.

VI. Examples

Mockus and Weiss [1] give an example of the application of chunking to identify chunks that were good candidates for globalization. Herbsleb and Mockus [2] measured the cycle time for a module of 257 files identified using this technique in an industrial system. The module had about four percent of the MRs crossing its boundary and, after adjusting for the number of files an MR touched, the number of releases affected, and whether or not the MR was created by the developer implementing the change, MRs crossing module boundaries took 50% longer to complete. Practical recommendations for the context of globally distributed development are in [3] and the code for the algorithm are in [4].

Acknowledgment

We thank those who have been working on providing additional examples of the application of the pattern, including Rachana Koneru and Jeff St. Clair.

References

[1] Mockus, Audris; Weiss, David; Globalization by Chunking: A Quantitative Approach, IEEE Software, March/April 2001

[2] James Herbsleb and Audris Mockus. Formulation and preliminary test of an empirical theory of coordination in software engineering. In 2003 International Conference on Foundations of Software Engineering, Helsinki, Finland, October 2003. ACM Press.

[3] Audris Mockus and David Weiss. Software chunks and distributed development. In Christof Ebert, editor, Global Software and IT: a Guide to Distributed Development, Projects, and Outsourcing, pages 69-82. Willey, 2011.

[4] http://mockus.org/chunking