Eliminate Navigation Code & Split Up God Class

Hayat BELLAFKIH

University of Mons

hayat.BELLAFKIH@student.umons.ac.be

April 9, 2016

Overview

Introduction

A systems reengineering pattern is a description of an expert solution to a common systems reengineering problem, including its name, context, and advantages and disadvantages.

Description

Law of Demeter:
The objective of Eliminate Navigation Code is to reduce the impact of changes by shifting responsibility down a chain of connected classes. The problem is these changes in the interfaces of a class will affect not only direct clients, but also all the indirect clients that navigate to reach it.

Solution

Iteratively move behavior defined by an indirect client to the container of the data on which it operates. The recipe for eliminating navigation code is to recursively Move Behavior Close to Data.

1. Identify the navigation code to move.
2. Apply Move Behavior Close to Data to remove one level of navigation.
3. Repeat, if necessary.

Deciding when to apply Eliminate Navigation Code can be difficult.

- indirect providers:
  - Each time a class changes, e.g., by modifying its internal representation or collaborators, not only its direct but also indirect client classes have to be changed.
  - Look for classes that contain a lot public attributes, accessor methods or methods returning as value attributes of the class.
  - Big aggregation hierarchies containing mostly data classes often play the role of indirect provider.
- indirect clients:
  - a sequence of attribute accesses.
  - a sequence of accessor method calls.

Split up a class with too many responsibilities into a number of smaller, cohesive classes.

- A god class monopolizes control of an application. Evolution of the application is difficult because nearly every change touches this class, and affects multiple responsibilities.
- It is difficult to understand the different abstractions that are intermixed in a god class. Most of the data of the multiple abstractions are accessed from different places.
- Identifying where to change a feature without impacting the other functionality or other objects in the system is difficult.
- It is nearly impossible to change a part of the behavior of a god class in a black-box way.

A god class may be recognized in various ways:

Detection

- a single huge class treats many other classes as data structures.
- a root class or other huge class has a name containing words like System, Subsystem, Manager, Driver, or Controller.
- changes to the system always result in changes to the same class.
- changes to the class are extremely difficult because you cannot identify which parts of the class they affect.
- reusing the class is nearly impossible because it covers too many design concerns.
- the class is a domain class holding the majority of attributes and methods of a system or subsystem.
- the class has an unrelated set of methods working on separated instance variables.
- the class requires long compile times, even for small modifications.
- the class is difficult to test due to the many responsibilities it assumes.

Incrementally redistribute the responsibilities of the god class either to its collaborating classes or to new classes that are pulled out the god class. When there is nothing left of the god class but a facade, remove or deprecate the facade.