Assignment 1

Object-Oriented Design, IV1350

Emil Tullstedt emiltu@kth.se

2014-05-19

Contents

1 Introduction

This report describes an analyze of the requirements for a cache layout and the process of modeling the analyze as a domain model and a system sequence diagram. The report is made for the course *IV1350* at KTH and is not made for usage in production.

While analyzing the application, I co-operated with *Martin Alge*, *Jesper Falk* and *Erik Pettersson*.

2 Method

There's a rather important difference between object-oriented design and object-oriented programming, similarly to the difference in the architect's process of designing a house and the artisan's process of building it.

The process of building a house is similar to the process of building software. For an artisan building a tree-house or raising a tent there's no need for a formal analysis, just as how simple applications and scripts often *just happen* for a programmer.

Whenever you're building anything slightly more advanced, ranging from a playhouse to a chateau - you probably want to do an analysis and a design before actually constructing. The artisan will probably find herself to be insufficiently experienced for building a chateau, and will almost certainly fail if she were to start building by herself. Just as any programmer almost certainly will fail trying to build anything as complicated as an usable operating system or word processor as a hobby project[1].

If the artisan were to try and construct a cottage, she would probably not need to consult an expert in the same way as the artisan would need to consult an expert in the process of building the chateau, but she would still have to spend quite a bit of time analyzing and designing the various plans for the cottage.

At the end of the day the process for doing any work which requires attention to both lots of details and the entirety is very much alike. Thus the artisan and the programmer has the same problems when facing a seemingly overwhelming project, and before implementation of the plans, all artists needs to consider the following:

1. What is the purpose of this project? (Analyzing)
2. What is needed to fill that purpose? (Designing)

This report briefly covers the process of a programmer analyzing a smaller software project following an object-oriented analysis and design formula using UML.

2.1 Domain Model

The very first step when analyzing a problem you've been tasked with solving programmatically would be to observe the nouns in the problem. When programming using the object-oriented paradigm, this is similar to the architects need of identifying what rooms to build and what furniture to place in each room. At this point the size of the bed doesn't really matter and it doesn't really matter if the kitchen has an inductive or a gas stove.

The purpose of the domain model in object-oriented analysis is to standardize the pattern of how a programmer shall scribble down the identified objects, much like how drawings have been standardized for engineers in general. This allows for a software architect to draw a domain model and then hand it over to a software implementer or a customer, and no matter

[1] Obviously, if the artisan happened to be Linus Torvalds, the artisan would find a bunch of friends and they would together build the grandest and most beautiful chateau, but that's quite a bit off topic.

which software architect have made the domain model, the customer or implementer should immediately be able to understand the model given that they are already accustomed to domain models.

3 Result

3.1 Domain Model: DataCache

When designing the domain model for the data cache (visible in fig. 3.1) the key nouns identified were:

- User
- Nickname
- Program
- Block Size
- Block Count
- Associativity
- Cache Layout
- Address Layout
- Address Tag
- Index
- Offset
- Cache
- Instruction
- Load
- Store
- Main Memory
- Rows
- Simulation data

When designing the domain model from this set of nouns it's important to embrace the idea that not all of these are actually conceptual classes in their own rights, for example, one can see that the *Cache Layout* is defined as a coalition of *Block Size*, *Block Count* and *Associativity*, just as the *Address Layout* is a coalition of the *Address Tag*, *Index* and *Offset*.

Based on this, you can build a category list where you set the nouns in different categories.

Category	*Content*
Actors	Program, User
Containers	Data Cache, Simulation Data, Main Memory, Rows
Contained items	Rows, Nickname, Address Tag
Operations	Instructions
Operation Types	Load, Store
Layouts	Cache Layout, Address Layout
Layout Properties	Associativity, Block Size, Block Count, Address Tag, Index, Offset

With the category list in place the process of eliminating nouns from being conceptual classes in the domain model begins. When doing so, first place everything that logically will be a conceptual class of it's own in the domain model to begin with, and then try to find how these correlates with each other. Objects that are of simple data types and only contained and doesn't have any calculations of it's own are quickly inserted as attributes of other objects, such as everything I put in the layout property category is the attribute of a layout-object (unsurprisingly). Iterate this process a few times, find connections and try to eliminate as many conceptual classes as possible without having objects that feels like huge monoliths.

There's a few choices made during this process in the domain model visible in fig. 3.1 that's worth highlighting, namely:

- The `Program` conceptual class is unnecessary, as it is simply a container for the entire execution of the application. The `System` in the system sequence diagram fig. 3.2 is roughly equivalent to this conceptual class
- The `Row` conceptual class is both a container and a contained object, and exists mostly because the instructions clearly says the `DataCache` may contain multiple locations
- The `Cache Layout` and `Address Layout` might be interesting to merge. However, there's quite a few operations relating to either of them there's a risk that including both within the same conceptual class will create a monolithic giant conceptual class
- The `Storage` conceptual class is based on the `Simulation Data` noun and the verb that requires the program to *store* key execution data
- The `User` conceptual class is the actor which operates on the program. The choice of storing nickname in the `Storage` conceptual class instead of within the user is because the user isn't conceptually seen as part of the actual programmable program, but as an external actor

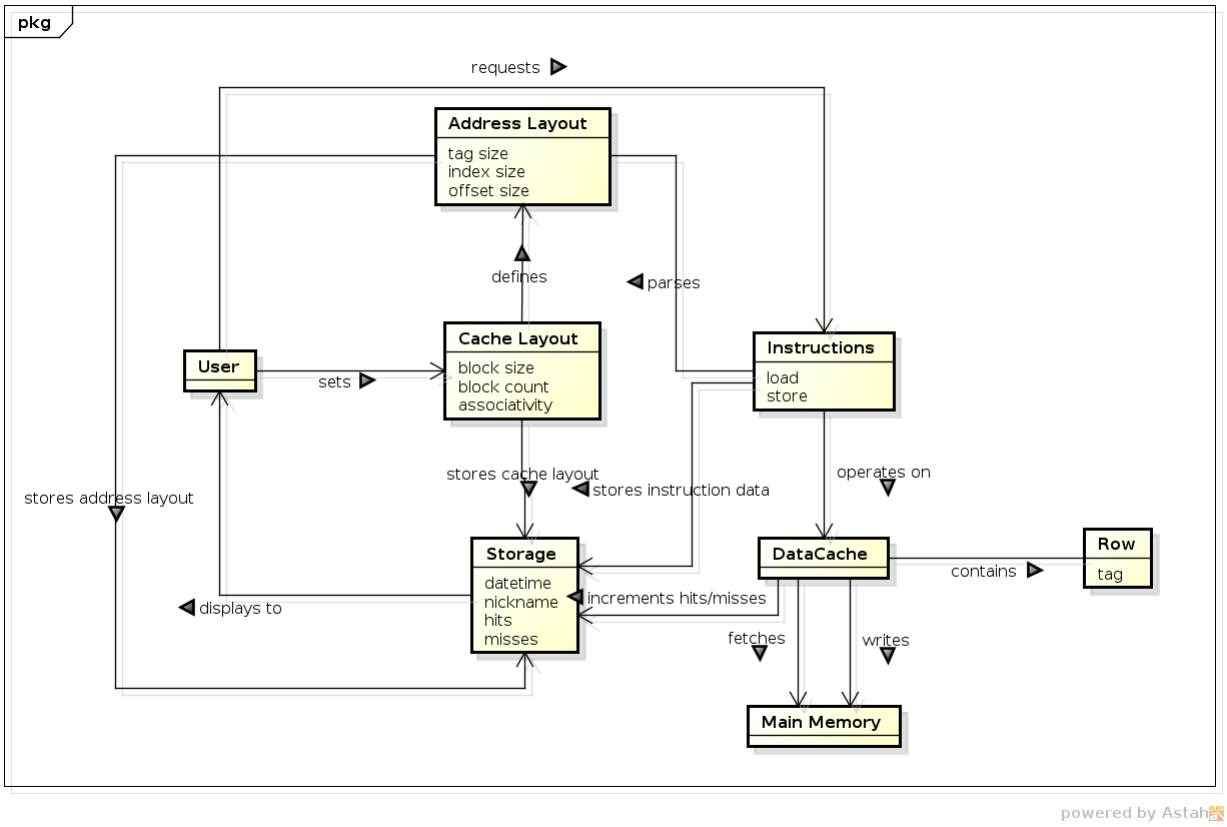

Figure 3.1: Domain model for the analyze of a data cache specified in the instructions which were made following the instructions in section 3.1

3.2 System Sequence Diagram: DataCache

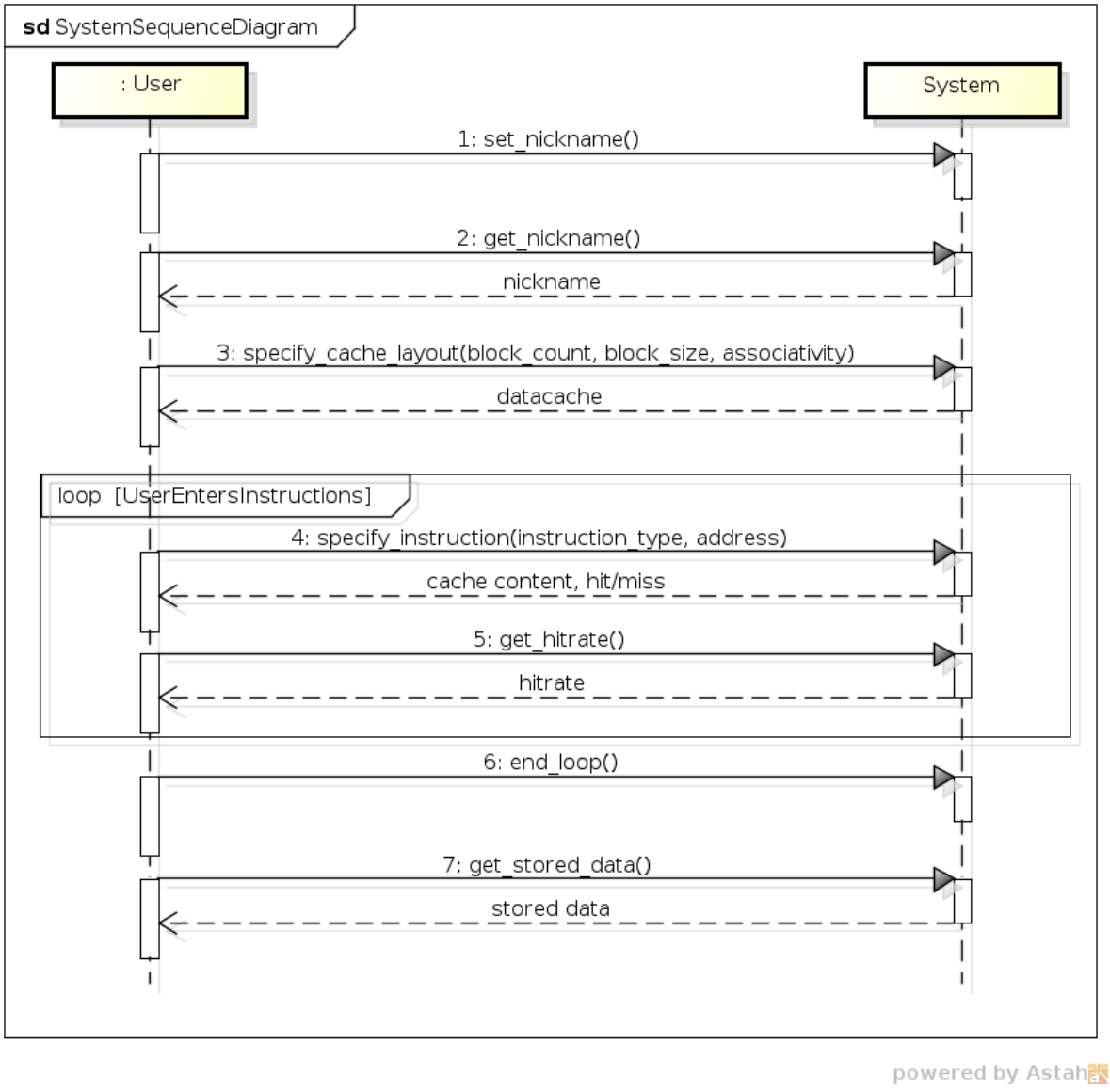

Figure 3.2: System Sequence Diagram for the data cache