**Subject Oriented Programming**

**Abstract**

This paper presents the way to design software using Subject Oriented Programming (SOP). SOP is basically programming with Objects, both real and abstract objects from first principle. The basic principle is that every object has only one behaviour (function or method). SOP comprises Objects, Grammar, idea density, generalization and factorization. These are the principles of SOP. Idea density is the principle that underpins Subject Oriented Programming. I call it Subject Oriented Programming (SOP) because most well-constructed English language sentences start with a subject (before predicate and then object) as it is inspired by English Grammar. I will use *Building and Using a House* to teach this new Subject Oriented Programming that I just invented.

**Introduction**

I will just gloss over Object Oriented Programming and go straight into Subject Oriented Programming (SOP) as OOP is well used already. SOP is programming software after real objects that we can feel. A good number of main-stream programming languages like Java, C++, etc. implemented OOP. We use Universal Modelling Language (UML) to design and IDEs to code. This include principles like SOLID and many more, but we are still not able to write maintainable, readable, and performing code. We are still not able to write decent data structures and algorithms as well. For some of us in the high-performance space, we find standard library nothing to write home about for any kind of use where performance matters. We normally resort to doing somethings differently as they concern data structures and algorithms, in order to get performance. I myself used to struggle with designing and writing code that performs really well. It goes without saying that software code must be readable, maintainable, robust, error free and really fast. Or let say fast enough for the Software Application. I mean software continue to be fast or even faster as Software Application load scales or increase. Hence, the emergence of SOP as the way to ensure that Software can have these qualities.

I think that the Software industry has battled long and hard with OOP so much so that its time the industry designed and programmed with Objects the proper way using SOP. The industry has gone through many bad design decisions after another in the life cycle of programming languages, from the use of null to piles of needless layers of abstractions such as Operating Systems, Virtual Machines, Cloud platforms and Containers. Well with the exception of OS. I think we should move a bit closer to the Turing machine as these layers of abstractions sometimes are unnecessary, which most often than not restrict our design freedom. I know some will be asking if functional programming is the answer. Functional programming is not exempted from these issues, even as it claims to be based on Mathematics. They clearly use virtual machines, which I believe are unnecessary. Besides, it has its peculiar problems too. I will discuss functional programming in my next paper.

I hereby claim that Subject Oriented Programming solves all of these issues nicely. I believe an obvious question is: if SOP solves all these problems or issues in developing software, what use are highly skilled software engineers or developers? I might as well replace every high skilled software engineer with mediocre ones. I promise there are other types of challenging problems for highly skilled software engineers such as myself and many others out there especially in the high-performance space. You will get a good sense of these challenging problems by the time you finish reading this paper.

Some of you will be dying to find out if there is a solution on how to implement SOP in a programming language. The answer is yes, but I am working on a pristine new compiler to demonstrate the ideas, but until I am satisfied with the compiler, which I will be working at for a while, the solution will still stay with me. Cheer up, I will publish the solution once the compiler is available on GitHub. That is the much I will say on that for now.

Sometimes, I miss the point of the conversation, and digress a little, but I have found it to help the circulatory system of the body. So, back to Subject Oriented Programming (SOP) since we have all warmed up a bit. What is Subject Oriented Programming? Subject Oriented Programming is the way to program with Objects. It is basically considering the meaning of the object I am programming in a sentence. Therefore, understanding of basic grammar of English language is key to SOP. I am here to explain how they all work as I go through, but let’s start with defining principles and concepts.

**Basic Principle of SOP**

Let’s start by asking a simple question: How can I design a house using **Subject Orientation**? First, I will consider the meaning of a house. What is a house? A house is an object that shelters the occupant. That’s it. This is the very basic principle of SOP.

Before you start venting anger, go and find something to drink. Martini vodka, if you will. I will settle for a coffee myself. While you are having your drink, a fundamental rule of SOP is that every Subject must have exactly one behaviour. I mean one method per Subject or Object. The only one exception is an adapter, which is rare and a class of its own. An Adapter can have at most and exactly two (2) methods. I will design adapter in my next paper. Apart from adapter you can be confident that every object has one method.

Now that we are all refreshed and ready to continue. Let me explain. Let’s consider the sentence: A house is an object that shelters the occupant. House is the subject of the sentence. ‘Shelters’ is the verb or action. Occupant is the object of the sentence. In essence the house performs an action on the object. This is natural language of objects in reality, analysed in a sentence. Let me put it in SOP: House.shelters(Occupant).

This is a recognizable piece of code, especially to those who are nurtured in Object Oriented Programming. Programming with objects is about sending messages to objects. In this context, house sends message ‘shelters’ to Occupant. Occupant is the object of the sentence, but normally occupant of a house is usually human being. One can argue, why not make the occupant the subject of the sentence and house which is object in real life the object of the sentence? Let’s just do that. Say: *shelter occupant house*. i.e. shelter(occupant).House. Now the action is directed to *house* and *house* is the object, but we don’t have subject anymore. We have predicate and object. In other words, we have message: shelter(occupant) and house that receives the message. Don’t panic, we still have subject, only that the subject is no longer explicit in the code, but the subject is the system, which actually sends the message to house. Typically, system sends messages in a computer. This clearly explains that message sending is implementation detail, which should be behind the code and rightly so. Therefore, message sending is not part of this paper, but I will publish it in another paper, after a working compiler is also available in Github. Besides, messaging sending is implementation detail that you wouldn’t want to get in the way of this paper.

There is one more variation to these. Say: let occupant be sheltered by house. Occupant.shelteredBy(House). This code gives you a sense that the prefix: sheltered of the actions is a state as opposed to being an action. Therefore, there is no message sent on this occasion, but it was sent in the past. This is an elegant way of defining and adding state to Subject in SOP. There we have it: state added to Subject as well.

Before going further let’s consider the model of SOP core syntax I will be using for the rest of this paper. House.shelters(Occupant) ‘shelters’ is an action as opposed to being a message. This clarifies that message sending is done behind the syntax and the system sends messages from one memory cell to another. This is what the syntax promises on the code itself, but the actual sending of message is done in the implementation of message sending model that lies behind the scenes. Evidently, the code (House.shelters(Occupant)) represents a message as a whole. This means that the subject does not send messages at all. The system does. The Subject directs action to the Object. The art of directing action to object represents the message being sent by the system.

Using the same model of syntax, can House exist without Occupant? Yes, it can.

I can have: House.shelters() and it will still be Subject Orientation. In fact, that is exactly why it called Subject Oriented Programming and not Object-Oriented Programming.

Without subject, nothing directs or performs action on the object, and nothing gets done at all. However, house perform action without directing to anything in particular. This means that whoever is interested can receive or take on the action. This is house services are created as well. Corollary, I can build a house and there is no occupant for it. This also implies that the house can get occupant anytime at some point. In software terms, it is the point of providing software as services (Hardware as service, Platform as service, etc.) You don’t direct these services to anyone in particular, you pay, and you get or use the services.

This feels a bit cheeky for those that actually write code. Subject that performs action but did not direct it to any Object in particular. It does not mean that no object actually receives it. It means that the Objects that will receive it are not explicit in the syntax. It just means that the programmer is free to choose what object it will be directed. This is where you are able to do any side effect you want: print to the screen, write to file, network or the disk, or all of the above. This is the basis of software services. It is also a command. How is it a command? The nature of command is that you give it to a subordinate and hope that he carries it out. There is no way of ensuring it is received at all let alone carrying it out. This is why the military have hierarchy and strict command chain that help enforce the carrying out of command by subordinate. In software terms, this code: House.shelter() represents a command – notice that the action is ‘shelter’ and not ‘shelters’. In essence, with ‘s’ it becomes services, without ‘s’ it becomes command. These things are not new to programmers by the way.

We are ready to code in Subject Oriented Programming. Relax, this is a universal design language.

There are many approaches to doing SOP and they are:

* Top down approach
* Bottom up approach
* Exploration approach

I am not going to cover all of them here in this paper, however I will be using different approaches for certain case studies in subsequent papers. Top down approach to SOP is basically starting from top-most layer of abstraction and design our way down to the finest detail. The reverse is the case with Bottom up approach: start from the finest detail I can think of and design my way up the top-most layer of abstraction. Exploration approach combines these two approaches as I design up and down layers of abstractions, generalizing as I see fit for the given problem. I will be using **Exploration design** **approach** for the following case study.

**Case Study: Building and Using a House**

I will be building a house with SOP. Basically, design a house, build it and use the house as we go through the use of SOP. Let’s continue coding in SOP:

House.shelters(Occupant)

We know that house shelters occupant from weather such as rain, snow, wind and cold. How does a house shelter? A house shelters with roof, walls and floor. Roof shields the weather from entering the house from the top. The walls shield the weather from entering the house from the sides and the Floor shields the weather from entering the house from the ground. The following SOP code represents these.

Roof.shields(Top)

Walls.shields(Sides)

Floor.shields(Ground)

We also know that:

Walls.sizes(Room)

Room.uses(Space)

As you can see, we are exploring what a house is and how it does what it does. And I am using the very basic syntax of SOP to design, there is no need for any diagram or any modelling language like UML for design in SOP. This is the way to design software. I mean the way to design and code any type of system. It is a ***general-purpose design language*** that can produce implementable **complete** design and as well actual working code for the software application being designed. SOP can help engineers, developers and Software Architects and Consultants to produce elegant software design. SOP can be used to design algorithms and data structures. Design code produced by SOP are guaranteed to perform well when implemented in a programming language. This is the way to design and code high performance computer programs. In theory any design and code produced by SOP will be maintainable, readable, robust, almost error free, and most importantly performs better than anything you can get from another means even in the same programming language. SOP is programming language agnostic, which means you can design with SOP and implement the code in any programming language of your choice. Design produced by SOP is guaranteed to perform better than what is produced by other design tools like Enterprise Architect/UML, etc.

Notice that Walls has two different methods in the code above. I will reconcile them as we design. The above code more or less presents the problem I am about to solve. Let’s design a house.

**Designing a House**

In order to design a house, we need an architect. Architecture or style of the house I want to design is very important to the endeavour. So, let’s start by saying Architecture (you could use architect instead, but I prefer architecture) designs structure - roof, walls, and floor. As a result, we get design document. Simply put the design of the house. The content of such document is what I am aiming to achieve with SOP, when I say design a house. So, let work towards achieving this objective. Don’t worry about duplicates and repetitions, they tend to be useful later in the design process.

Architecture.designs(structure): Design

Architecture.designs(Walls/Roof/Floor): Design

Walls.shape(Room): Space

Room.uses(Space)

Space.takes(Bed/Sofa/Table/Coker/Tub)

Space.takes(Equipment)

Space.takes(Furnishings)

Equipment.services(Occupant)

Structure.shapes(House/Room)

Walls have some ambiguity associated to it, as house can have internal and external walls, but I don’t need to border about it yet until there is need to differentiate them. So, we are fine. As I explore, I repeated space as *subject* three times, that is because I don’t know which one will be needed or will help in different way. Nevertheless, I like to keep my design focused.

House.shelters(Occupant)

Architecture.designs(structure): Design

Walls.shape(Room): Space

Room.uses(Space, Furnishings)

Furnishings.fitsIn(Room)

Occupant.uses(Furnishings)

Structure.shapes(House/Room)

Observe that *Space* has been used differently with *Furnishings* as objects in *Room* which is *Subject*. It makes sense and immediately simplifies the design. *Room* uses *Space* for *Furnishings*. *Furnishings* fits in *Room*, and *Occupant* uses *Furnishings*.

House.shelters(Occupant)

Architecture.designs(Structure): Design

Walls.shape(Room): Space

Area.dimensions(Room): Space

Room.uses(Space, Furnishings)

Room.fits(Furnishings)

Occupant.uses(Furnishings)

Structure.shapes(House/Room)

Observe that *Area* is a member of *Walls*, hence the indentation. Room could fit furnishings i.e. *Room* fits *Furnishings*. So far so good but we are not there yet. Let’s try to assign responsibilities to the subjects and objects as much as we can. As you must have observed each line of code represents a complete sentence. I am sure you are making sense of what I am doing as I design by reading SOP code.

House.shelters(Occupant)

Architecture.designs(Structure): Design

Walls.shape(Room): Space

Area.dimensions(Room): Space

Room.fits(Furnishings)

Occupant.uses(Furnishings)

Structure.shapes(House/Room)

Observe that *Room* fits *Furnishings* has been chosen instead of the one that uses space and furnishings as objects. Why? Because if room fits furnishings it has used space for furnishings, simple. Notice also that *Structure* can shape *House* or *Room*. One room house is still a house. This defines *interface* as it is used in modern programming languages. I have just produced a house design. Now let’s build the design.

House.shelters(Occupant)

Architecture.designs(Structure): Design

Structure.[Interior: Morden, Exterior: Gothic]

Area.dimensions(House)

House.[size, roomCount]

Area.dimensions(Room)

Room.[length, width]

Project.builds(Design): Building

Design.lays(Bricks): Walls

Walls.shape(Rooms): Building

Structure.shapes(House/Room)

Building.furnishes(Rooms)

Room.fits(Furnishings)

Occupant.uses(Furnishings)

I elaborated on the House design, by specifying some objects’ states. I just did enough to capture the essence or the purpose of the design. Observing this rule keeps my design complete for the purpose it is created. Of course, I can add or remove from the design as I see fit and the resulting design will still be complete. I will show how this can be achieved as I apply the principles of SOP going forward. Next is building the house design that has just been produced. The last piece of code takes us to the *Using the House.* Notice also that I did not have to disambiguate Walls (internal and external), as the purpose of the design does not necessitate doing that just yet.

**Using the House**

Let’s consider how the house can be used and continue our exploration. How you explore the uses of the house depends on personal taste, but here is how the house can be used.

Occupant.uses(Furnishings)

Occupant.uses(BedRoom)

Bed.relaxes(Occupant)

Room.fits(Bed)

Occupant.uses(LivingRoom)

Occupant.uses(SittingRoom)

Sofa.relaxes(Occupant)

Room.fits(Sofa)

Occupant.uses(DinningRoom)

DinningTable.dines(Occupant)

Room.fits(DinningTable)

Occupant.uses(Kitchen)

Occupant.uses(Cooker)

Room.fits(Cooker)

Cooker.serves(Occupant)

Occupant.uses(BathRoom)

Bath.washes(Occupant)

Shower.washes(Occupant)

Room.fits(Shower)

Room.fits(Bath)

I am sure you observed how fluent design can be with SOP. From here going forward I will start applying the use of ***Idea Density*** to the design of the house, which makes SOP a fantastic tool to design anything you want in Software. It’s all about English language and mathematics. Furthermore, we can see that Occupant uses a lot of thing in the code above. This means that *uses* in *Occupant* will be complicated and clunky too. This is a design smell. Let’s also change *Occupant* to *Person* for simplicity.

Room.recieves(Person)

Bed.relaxes(Person)

Furniture.supports(Person)

‘Basically, supports person’s torso’

This code above says that we are designing a bedroom and bed is a piece of furniture that can support person, more specifically a person’s torso. I know this looks like the sort of indentation in JavaScript that no one likes. But it makes things clear as long as you keep it to only three levels, which is the standard in SOP. Alternatively, I can align the code as follows:

Room.recieves(Person)

Bed.relaxes(Person)

Furniture.supports(Person)

‘Basically, supports person’s torso’

The two communicates the same things only that the structures are different. This is the meaning of refactoring in test driving development. You can change the structure of code but not change the functionality of the code, which is what we just done. Observe, also that person using the room, and bed does not have any functionality about the room, bed and furniture. This is a better design. Let’s continue coding.

Room.recieves(Person)

Sofa.relaxes(Person)

Furniture.supports(Person)

‘Basically, supports person’s torso’

Room.recives(Person)

Dining.dines(Person, Food)

Furniture.support(Person)

‘Basically, supports person’s torso’

Room.recieves(Person)

Bath.washes(Person)

Shower.washes(Person)

Observe that the dining room and diner have two objects, which are basically two arguments, which means that room has two behaviours, even though it should have just one behaviour. This is one of design conundrums that idea density is for, but I will show how it can be solved in the process of this design.

Let’s select a room and consider in dept since I repeated the use of it in many ways. Before going into that let resolve the issue with dining room and other rooms, using idea density. Dining receives person and food. When person enters a room, he enters for a purpose hence the room names. For example. I enter the bedroom to lie in the bed, I enter the dining to eat food. As I enter these rooms I entered as person. There is no indication of what I entered with, I could have entered naked or with a bag, etc. These details were not mentioned because it is not relevant to the purpose of the design so far. I had you going there haven’t I. Let’s go back to dining. I can enter the dining as the master of the house to eat and a servant enters to serve food in the dining, so both are persons. Therefore, servant and master are all persons, but they capture different things or connotations. So, I can reconcile these ideas about entering a room into a denser form, which is still person. This is normally implemented as interface which discovered earlier, and it also exists in modern programming languages. Let’s go ahead and consider the leaving room.

Room.recieves(Person)

Sofa.relaxes(Person)

Furniture.supports(Person)

‘Basically, supports person’s torso’

Let’s consider the Sofa and Furniture with idea density. Sofa provides relaxation service and so does bed. I can sit in a sofa with my bottom and torso relaxed, but I can lie in bed with my torso, bottom and legs relaxed. Moreover, If I had lazy boy sofa, I can actually relax my torso, bottom and legs as well. This actually means I can generalise bed and sofa, but not so fast yet. Let’s see how dense this idea runs. When I enter the dining to eat, I actually sit on a chair, which supports my bottom and torso at same time, sitting upright. It can as well support the legs if the chair has footrest, but let’s leave it out as it is not general enough for our purpose of design. So still keeping the common denominator as bottom and torso, which is basically the body. Having understood all these by the way of idea density, let’s look at furniture and sofa. I know you are eager to see where this is going, wait for it. Sofa is actually a piece of furniture, hence why furniture is a member or state in Sofa. This is called inheritance by composition, not by extension. This is actually the way to do inheritance. I will come back to this shortly. But Sofa is a type of furniture that provides seaters for person. But I can put a book on a sofa seat, and it will be a genuine good or proper use of sofa. This means that I can relate to sofa as just a surface in this context. This is demonstrating the power of idea density in SOP. Therefore, the code can look like the following:

Room.recieves(Person)

Surface.supports(Person)

This is looking like a thing of beauty, isn’t it? I am just beginning to design the use of room. So, fasten your seat belt let me give you some treat with what idea density can do maybe by tomorrow I will flex the full muscle of idea density, but for now let’s just treat you. In every room person can enter and as well leave the room. Since we are working on leaving room let’s say person enters the room to sit, if the seats are all taken or occupied. Person has to stand. I know capacity of the room and seats are important but let’s do baby steps for now.

Room.recieves(Person, IN/OUT)

In[SIT,SIT,STAND,STAND], Out[], Person[P1,P2,P3,P4]

Registers(In[],Out):cursor

Surface.supports(Person, SIT/STAND)

Capacity[]

Update(STAND/SIT)

This is beginning to look like serious coding; yes, indeed it is serious coding and yet design like you never seen it before. Now the room has a way of indicating that person has entered the room or just left the room by IN/OUT flag. It also keeps track of who is in the room by number of people standing and sitting. Wait, standing is supported by floor, which is surface, so I don’t have to do anything more to support it in the code. Because it actually supports the entire body, so we might as well use body and person interchangeably, but for now I am fine with person. I will expand the code a bit more before I start simplification, but Let’s do something different with Idea density before coming back to this.

Let’s consider the kitchen and whip out some food really quickly. It’s all about idea density still.

Room.recieves(Person): Food

Person.prepares(Ingridents): Recipe

Cooker.cooks(Recipe): Food

Let me peel the name kitchen off the room and put the meaning of kitchen in the room instead. You must have observed the same principle applied to Sitting Room and Bedroom earlier. But how can we ensure that only chef can prepare or prepares food?

Room.recieves(Person): Food

Person.gets(Ingredients): Recipe

Chef.prepares(Ingredients): Recipe

Cooker.cooks(Ingredients, Recipe): Food

I think person should get ingredients, then chef prepares the ingredients and the cooker cooks the ingredients with recipe. I am not done yet. As a person can be chef, engineer, doctor, dad, etc. This means a person can put on as many hats as they can learn or have skills for. But how can person choose a hat to wear. Let’s focus on chef and surgeon for a moment. Let’s explore what a surgeon is. A surgeon operates on human beings or living things in order to cure ailment or disease. They usually perform surgery in theatres. But in order for a surgeon to perform surgery he has to adhere to a well laid down rules or procedure from start to finish of the operation. This is beginning to feel like recipe for chef. In a nutshell, a chef uses a recipe to prepare or cook dish and surgeon uses surgical procedure to perform surgery. It goes without saying that there are many recipes for different dishes and different procedures for different surgeries. Apparently, I am defining skill, which simply means a set of instructions for a specific job. Using this set of instructions is what surgeons and chefs do, on the other hand creating them is what master chefs, consultant surgeons and consultant engineers do. However, the generalization of both sets of professionals is skills. Let’s do some coding

Person.mixes(Ingredients): Recipe

PERSON.tries(Ingredients): Order

Ingredients.documents(Order): Recipe

Order.sorts(Ingredients): Recipe

‘In place sorting of ingredients’

This is a bit complicated, so let’s just assume that the recipe has been created.

Room.recieves(Person): Dish

Person.prepares(Ingredients, Recipe): Dish

Recipe.sorts(Ingredients)

Cooker.cooks(Ingredients): Food + Plate = Dish

Room.recieves(Person): OutPatient

Person.operates(Patient, Procedure): OutPatient

Procedure.Operates(Patient): OutPatient

I can generalize as I see fit by finding common denominators. Recipe and procedure represent skill. Patient and ingredients represent the work or rather profession, but ‘operates’ is more general, even though it is used in parochial contexts, but definitely better abstraction to use here, so I will make changes to the code

Room.recieves(Person): Dish

Person.operates(Profession, Skill)

Skill.manages(Profession)

Cooker.cooks(Profession): Dish

Room.recieves(Person): OutPatient

Person.operates(Profession, Skill)

Skill.manages(Profession): OutPatient

So that’s it in a nutshell. This is the magic of idea density, but it is yet to flex its muscle. Stay tuned. Let’s stick with the kitchen for a little longer. Dish implies plate and food, so dish is a higher abstraction to food. The code I have also implies that Person does not have any skill or profession, they are injected given context. How can I make person have skills or profession? By craft. Craft basically involves *skills* and *handling*. This means that skill can replace profession and recipe takes its place, for now let’s keep the handling part at the back burner. This is idea density still. I now have the following code:

Room.recieves(Person): Dish

Person.operates(Skill, Recipe)

Recipe.manages(Skill)

Cooker.cooks(Skill): Dish

Recipe is basically a document of list of instructions/procedure. Procedure seems more general, so let’s use procedure instead of Recipe. Remember also that I have unused handling above, so mathematically it replaces the action just before Skills, which is operates. i.e. handles is ***used*** instead of operates. But there is also another Skill in another context in the next piece of code, so I replace action: manages with handles instead and replace operates with uses – just discovered. Now I have the following code:

Room.recieves(Person): Dish

Person.uses(Procedure): NewProcedure

Skill.handles(Procedure): NewProcedure

Cooker.cooks(Ingredients, NewProcedure): Dish

Idea density to the rescue. Let’s keep designing.

Room.recieves(Person): Dish

Person.uses(Procedure): NewProcedure

Skill.handles(Procedure): NewProcedure

NewProcedure.cooks(Ingredients): Dish

Cooker.heats(Ingredients): Food

Plate.dishes(Food): Dish

How does procedure know the matching skill? Normally in software programs, I can use ‘If’ statement to find the matching skill, but before that let’s see if I can design out the use of ‘if’ statement. Another way of doing this is to use Closure or Lambda expression in software engineering. But this is easy to do, I have done it already, just inject the code that represent the skill you want along with the procedure and that’s it. But let’s go by the way of design. Let’s use pattern matching for this, which means I have to assume that there is an array of skills in person. It involves having a way of identifying the skills in the given procedure and then selecting relevant skill by dead reckoning. Let’s focus on Person:

Person.uses(Procedure): NewProcedure

Chef.handles(Procedure): NewProcedure

Order.sort(Procedure): NewProcedure

Person.uses(Procedure)

Order.sort(Procedure)

This is an elegant design. However, Let’s warm up for the actual flexing of SOP muscle. Let’s continue with where I left off with the Sitting room. Let’s refresh my mind with the code.

Room.recieves(Person, IN/OUT)

In[VCT,VCT,VCT,VCT],Out[SIT,SIT,STAND,STAND, Person[P1,P2,P3,P4]

registers(Person, Person[])| delete(Person, Person[])

registers(In[],Out):cursor | register(Out, In)

Surface.supports(Person, SIT/STAND)| same

Capacity[]

Update(STAND/SIT/VACANT)

This is the initial state of the method. As the main purpose of entering the room is to sit, there the seats will be taken before the standing positions. The first *registers* records the person that entered the room. The second *registers* records the availability of seats and stands and occupied ones. The surface keeps track of the sitting or standing on the actual infrastructure. This is just entering the house. Let’s code leaving the room alongside the entering code. Registering and person and deleting a person when leaving, suggest that registers is a actually creates and I can make that piece of code denser by abstracting both to updates a person. The register in and register out suggest symmetry, so the code can be denser in form of registers. The surface code is same

Room.recieves(Person, IN/OUT)

In[VCT,VCT,VCT,VCT],Out[SIT,SIT,STAND,STAND, Person[P1,P2,P3,P4]

updates(Person, Person[], FWD/BWD)

registers(In[],Out, FWD/BWD):cursor

Surface.supports(SIT/STAND/VACANT, cursor)

Capacity[]

Update(STAND/SIT/VACANT, cursor)

The application of idea density on these methods introduced backword and forward of the cursor. This essentially means that forward move is entering, and backward move is leaving. Let’s elaborate on the first updates: and registers as follow:

Room.recieves(Person, IN/OUT)

In[VCT,VCT,VCT,VCT],Out[SIT,SIT,STAND,STAND, Person[P1,P2,P3,P4],cursor

updates(Person, Person[], FWD/BWD)

put(Person, cursor, FWD/BWD)

move(cursor, FWD/BWD)

registers(In[],Out, cursor FWD/BWD)

vout = get(OUT[],cursor) | vin = get(IN[],cursor)

vin = get(IN[], cursor) | vout= get(OUT[],cursor)

put(vin, OUT[], cursor) | put(vout, IN[],cursor)

put(vout, IN[], cursor) | put(vin, OUT[],cursor)

Surface.supports(SIT/STAND/VACANT, cursor)

Capacity[]

Update(STAND/SIT/VACANT, cursor)

Observe clear symmetry in registers. I do the same thing irrespective of whether I am entering or leaving. By the way code symmetry is a rear phenomenon. I will take a moment to savour the symmetry. Let’s apply idea density to it and move forward. I will compress to get a denser code as follow:

Room.recieves(Person, IN/OUT)

In[VCT,VCT,VCT,VCT],Out[SIT,SIT,STAND,STAND, Person[P1,P2,P3,P4],cursor

updates(Person, Person[], FWD/BWD)

put(Person, cursor, FWD/BWD)

move(cursor, FWD/BWD)

registers(In[],Out, cursor FWD/BWD)

get(In[],Out[],IN/OUT, cursor): VALUE

put(In[],Out[],IN/OUT, cursor)

Surface.supports(SIT/STAND/VACANT, cursor)

Capacity[]

Update(STAND/SIT/VACANT, cursor)

Let’s take them one at a time. First up updates. How can I generalize the updates properly? It will help to have clarity of what’s going in detail as follow:

IN: put(Person), cursor = cursor + 1 FWD

OUT: put( ) , cursor = cursor – 1 BWD

Generalizing further with the most basic of arithmetic: addition and subtraction.

put(Person), cursor = cursor + 1 FWD

+ put( ) , cursor = cursor – 1 BWD

-------------------------------------------------------------------

put(Person), cursor = cursor [idempotent]

This means that when person enter and leaves the cursor does not change at all.

However, when I use subtraction, I get a different result as follow:

put(Person), cursor = cursor + 1 FWD

- put( ) , cursor = cursor – 1 BWD

-------------------------------------------------------------------

Person, -1 [-1 -1 = -1 {idempotent= for now}]

For now, I will take forward the result of the addition and come back for the result of the subtraction later. Before that, looking at the register code, it simply means swapping what’s in IN[] with OUT[] and nothing more, so I will generalize as such. To add the effect of addition result I will append move at the end of the ‘receives’ method. This means there is no need to keep records of persons and as well person in the method argument. The update in the Surface will need only ‘put’. Since we are done with simplification at this level, I will remove the ‘move’ in the beginning and end of the ‘receives’ method

Room.recieves(IN/OUT)

In[VCT,VCT,VCT,VCT], Out[SIT,SIT,STAND,STAND], cursor

swap(In[],Out[], cursor)

Surface.supports(SIT/STAND/VACANT, cursor)

Capacity[]

put(STAND/SIT/VACANT, Capacity[] cursor)

Let’s say that [SIT, SIT, STAND, STAND] = Services and services.Pos accesses the content of services that I want i.e. services.0 = SIT, services.2 = STAND, etc. and that Services are immutable. This way I can keep the method invariant outside the room. Clearly I can see that Surface is not actually need as I have everything need in the IN[] and OUT[]. Therefore, I can have the following code:

Room.recieves(Services)

Cursor = Services.pos

swap(In[],Out[], cursor)

Let’s make it even more elegant.

Room.recieves(Services)

swap(In[],Out[], Service.pos)

The room invariant is safely kept outside of the room. Having done that let’s have another look at the ‘updates’ method.

updates(Person, Person[], FWD/BWD)

put(Person, Person[], cursor, FWD/BWD)

move(cursor, FWD/BWD)

Let’s explore the idea density of actions: *put* and *move*. I have done that quite a bit but let’s explain it. What is ‘put’? put means place something somewhere. This in turn say I have to have to something and where to place it. Moreover, put relates to move in this context as putting data involves moving to where to put it. What is move? Move relates to change, and if I have change, then I have a reference point from where to change to a different state or position. In a nutshell, move means direction and something change position. It also implies something changing position and if position must be changed, I have to have a direction to which to move. Let’s illustrate this with an array of given capacity.

-1 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9].

Moving the cursor from index -1 to 0 represents the creation or addition of an arbitrary object. Moving from index 0 to 1 adds another arbitrary object, and so on. Moving the cursor in the opposite direction means deleting an arbitrary object, one move, one object deleted. This means that I don’t need ‘put’ anymore to implement ‘update’. But I need capacity or the size of the data to create or delete. On the other hand, cursor is pointer or counter. Therefore, I have the following code:

updates(FWD/BWD)

move(cursor, capacity, FWD/BWD)

The forward and backward moves are straight forward, just add one or subtract one accordingly, but what happens when the pointer is on -1 and I made a backward move? The cursor = -1 - 1 = 0, The cursor has just moved forward (creation) and not the expected backward (deletion). Let’s do both forward and backward. i.e. cursor = -1 + 1 - 1 = -1. This means no creation or deletion, which is better, but it is best not to allow any backward movement when the cursor is on -1. Similar issue occurs when the cursor is at the capacity and I want to do forward movement. Let’s explore the forward and backward movement with capacity of 10.

[(-1, 9), (0, 8), (1, 7), (2, 6), (3, 5), (4, 4), (5, 3), (6, 2), (7, 1)]

[(1, 7), (2, 6), (3, 5), (4, 4), (5, 3), (6, 2), (7, 1), (8, 0)]

[(0, 8), (1, 7), (2, 6), (3, 5), (4, 4), (5, 3), (6, 2), (7, 1)]

This illustrates how to move forward and backward with a cursor at same time. Observe the wrap arounds at the edges and none them allows moving out of the capacity of moving into negative one after moving out it initially and very importantly the full capacity is being used and traversed as well. Notice that I am able to move back and forth (-1, 9) to (0, 8), normally.

(-1, 9) : (-1 + 1, 9 – 1) to (0, 8).

(0, 8) : (0 – 1, 8 + 1) to (-1, 9).

I can prove that this is the case as well:

-1 +1 - 1 = -1

-1 -1 + 1 = 1 (not -1)

0 - 1 + 1 = 0

0 +1 - 1 = 0

Simply put it is wrong to say that -1 - 1 = -2. I have a complete and comprehensive proof that this is absolutely wrong. I also have proof that -1 - 1 = 0 is absolutely correct. Based on my correct mathematic, I am able to count forward from negative one and wrap around the edges of the capacity and zero without going into negative again.

Direction makes move a vector quantity, but if there is no move there is a probability of reading instead. Therefore, reading is a scala quantity. Direction shows whether move should be up, down forward or backward, but let’s stick with forward and backward. It is also important to know that every move forward or backward intrinsically has a read property which is a scala. This means that every update is a read as well. Read is also the property that can exist without other properties: create and delete. However, it exists only once and it is the initial condition of the array, if and only if the initial condition is not negative one (-1). This initial condition is the only one-off chance to read without any movement at all. The initial state can be chosen, but after the initial state every read is preceded by a move. Let’s modify the ‘update’ code to reflect every condition and logic when traversing a given capacity.

updates(CREAT/DELETE/READ, FWD/BWD)

if (CREATE && fwd < capacity)

move(FWD); put(p); read()

elseif(DELETE && bwd > -1)

move(BWD); read()

elseif(bwd > -1 && fwd < capacity)

read()

elseif(bwd == -1)

move(FWD); put(p); read()

elseif(fwd == capacity)

mov(BWD); read()

elseif(bwd < -1 || fwd > capacity)

do nothing

* Applying Exploration
* Grammar and Sentence
* Abstraction/Generalizing
* Deduction/Extraction
* Factoring Meaning
* Top Generalization
* Bottom Generalization

Conclusion

Reflection

Owara