

Software Developer's JOURNAL

new ideas & solutions for professional programmers

Vol2 No.02 Issue 02/2014 (7) ISSN 1734-3933

OPEN

THE LATEST INNOVATIVE METHODS IN PROGRAMMING

PLAY-BY-POST RPGS ARE ALIVE AND WELL
USER ACTIONS AROUND MVW
LANGUAGES IN ULS

TECHNICAL INTERVIEWING TECHNIQUE: LOOKING FOR AN INTUITIVE NARRATIVE
BY SOUMEN SARKAR JEFF EDMONDS

THE LATEST INNOVTIVE METHODS IN PROGRAMMING

Copyright © 2014 Hakin9 Media Sp. z o.o. SK

Table of Contents

Play-by-post RPGs are Alive and Well <i>by Alexander Hinkley</i>	05
User Actions Around MVW – Part 1 <i>by Damian Czernous</i>	09
User Actions Around MVW – Part 2 Associations <i>by Damian Czernous</i>	16
CMS-Based Web Application Maintenance Made Easy with Integrated OO Design <i>by Jean-Pierre Norguet</i>	21
Brand Integrity With Effective DevOps <i>by John Marx with Cigna and Capital One</i>	34
Actualizing The Potential Shippable Increment <i>by John Marx</i>	35
Languages in UIs <i>by Damian Czernous</i>	38
Design Patterns in Perl – Part 1 <i>by Pravin Kumar Sinha</i>	45
Design Patterns in Perl – Part 2 <i>by Pravin Kumar Sinha</i>	56
Design Patterns in Perl – Part 3 <i>by Pravin Kumar Sinha</i>	78
Technical Interviewing Technique: Looking for an Intuitive Narrative <i>by Soumen Sarkar Jeff Edmonds</i>	137
A Natural Programming Method. Programming with Natural Language <i>by Tsun-Huai Soo</i>	143

Hello Software Developer's Journal Readers,

Welcome to our first released issue...

SDJ magazine team pleases to announce launching the first issue of the free Open magazine. In this issue, a lot of tutorials and practice rich articles are embedded for you to develop your SDJ skills and knowledge. Our ultimate goal is to provide our readers with exactly the knowledge and skills they need in their IT careers. Hence, we will be very glad to receive your suggestions of workshops, tutorials, what you need most, etc...

Let's take a look at what you will engage in this free issue, Our experts will teach you the fundamental design patterns in Perl. In addition, you will discover the languages in UI and how to maintain your App localization and reusability. Additionally, you will learn how to improve your mobile product lifecycle and more of other content-rich articles.

We wish to say "Thank You" and express our gratitude to our experts who contributed to this issue and our coming workshops, however, we invite other experts for collaboration for the next issue, due in 4 weeks.

Stay Tuned, along the whole summer, we were preparing a set of practical workshops for you to be released this month.

Python Web Development: Our consultant, J. Tynan Burke, in this workshop, will teach students the basics and the finer points of web templating, using libraries like Boto for handling static files with Amazon S3, using services like Heroku to maximize efficiency and more.

iOS8/swift programming: Zhou Yangbo, our technical expert, assists readers in learning how to use Swift and SpriteKit to programming, AI-Steering Behaviors, use Swift to programming an Vector2D class, create games in Swift and **a lot more**.

R programming: Our instructor, Jim Lemon, introduces all about the R programming, fundamentals, functions, base R statistics, R graphics and more.

IF interested in getting real life technical experiences with our rich content SDJ workshops, ssues, tutorials, etc., Or want to get in touch with our team, please feel free to contact Gurkan Mercan at "gurkan.mercan@bsdmag.org", **Contact us TODAY and gain a -14 day- free access to all our workshops and issues. OR HURRY UP and contact us this WEEK and enjoy our limited annual offer of subscription for only 300\$.**

Hope you enjoy the issue.

Slawek Szeremeta
slawek.szeremeta@sdjournal.org



Editor in Chief: Slawek Szeremeta
slawek.szeremeta@sdjournal.org

Editorial Advisory Board: Shahid H Rathore, Craig Thornton,
Hani Ragab, Kishore P V

Special thanks to our Beta testers and Proofreaders who helped us with this issue. Our magazine would not exist without your assistance and expertise.

Publisher: Paweł Marciniak

Managing Director: Ewa Dudzic

Production Director: Andrzej Kuca
andrzej.kuca@sdjournal.org

Art. Director: Ireneusz Pogroszewski
ireneusz.pogroszewski@sdjournal.org
DTP: Ireneusz Pogroszewski

Marketing Director: Ewa Dudzic

Publisher: Software Media SK
02-676 Warsaw, Poland
Postępu 17D
email: *en@sdjournal.org*
website: *http://sdjournal.org/*

Whilst every effort has been made to ensure the highest quality of the magazine, the editors make no warranty, expressed or implied, concerning the results of the content's usage. All trademarks presented in the magazine were used for informative purposes only.

All rights to trademarks presented in the magazine are reserved by the companies which own them.

DISCLAIMER!

The techniques described in our magazine may be used in private, local networks only. The editors hold no responsibility for the misuse of the techniques presented or any data loss.



[GEEKED AT BIRTH]



**You can talk the talk.
Can you walk the walk?**

[IT'S IN YOUR DNA]

LEARN:

Advancing Computer Science
Artificial Life Programming
Digital Media
Digital Video
Enterprise Software Development
Game Art and Animation
Game Design
Game Programming
Human-Computer Interaction
Network Engineering
Network Security
Open Source Technologies
Robotics and Embedded Systems
Serious Game and Simulation
Strategic Technology Development
Technology Forensics
Technology Product Design
Technology Studies
Virtual Modeling and Design
Web and Social Media Technologies

www.uat.edu > 877.UAT.GEEK

Please see www.uat.edu/fastfacts for the latest information about degree program performance, placement and costs.

Play-by-post RPGs are Alive and Well

by Alexander Hinkley






Play-by-post role playing games (PbPRPGs) are very easy for indie developers to create. Unlike graphical RPGs or Massively Multiplayer Online RPGs, PbP games do not require any programming knowledge or coding experience to create other than perhaps some HTML for working on a website and a firm grasp of RPG mechanics. Play-by-post games instead utilize forum and/or chat services like instant messenger and players progress character storylines or perform in-game actions through collaborative creative writing. PbPRPGs were especially popular in the late 1990's and early 2000's. Their popularity has declined in recent years as gamers have become more enthralled with other – free to play – games such as League of Legends, but PbPRPGs still have a healthy online presence due in large part to how simple they are to make. The fact, pretty much anybody can start up their own PbPRPG means there are literally thousands of active PbPRPGs online at any given moment.

Play-by-post RPGs may be easy for an indie developer to create, but the creation process is still very complex. The first thing you need to do is to determine the theme you want your game to be set in. *Will it be a fantasy game or will it be a science fiction?, Will it be set in modern days or at some point in the past?* RPGs centered around specific fandoms usually have the most success opposed to an RPG that is a completely unique world. For example, RPGs that are set in the world of *Dragon Ball Z* remain one of the most popular genres for this type of game online. The reason that fandom RPGs work the best is because you already have a good chunk of your marketing done before you even wrote the first line of HTML for the site. When you create an RPG based on a fandom, people that are fans of that series at some point are going to search online for games to play set within that universe.

With a little search engine optimization, your site can rank among the top search results (e.g. Someone who is searching “DBZ RPG”) which will help you gain more members and exposure. Compare this with a site that takes place in a unique world that you created and has nothing to do with an existing intellectual property. Not only you will have to invest time and probably money in heavy advertising so people can actually find your game, but also you need to convince potential members that it is worth their time to sign up and play. This isn't an obstacle for fandom RPGs because those fans of the series *already* want to play. One thing that you will need to keep in mind if you chose to create an RPG based on a fandom, is that you aren't going to be able to monetize the site whatsoever because you don't own that intellectual property.

If you're a small play-by-post RPG with a few dozen members, there are chances that nobody is going to care about your site. But if you start getting a few hundred members and high levels of traffic, it might be a good idea to contact the company who owns that IP and ask for permission to keep running the site. This is the biggest disadvantage of creating an RPG based on an existing series, but it can be worth it if you're tapping into a huge audience of fans.

Once you have made up your mind on what theme you want for your PbPRPG, you have to determine what level of roleplaying you want the game to have. This is called the style. Threads on a play-by-post RPG can range anywhere from just a few sentences per post to thousands of words per post. Although the latter is generally thought of as taking more “skill,” the former can also be a lot of fun because it encourages highly interactive, fast-paced posting. When determining the style of your site, consider the age group and education level of your target audience. If your site is aimed at younger kids, for example, they probably aren't going to be interested in writing up long essay-like posts so a shorter role-playing style would be more suitable. Longer roleplays fit fantasy RPGs well because most fantasy novels are pretty lengthy (looking at you, *Wheel of Time* series). Fantasy fans are used to reading epic books with intricately detailed descriptions and will want to emulate that writing style in their role plays.

• OOC Boards				
	Forum	Topics	Replies	Last Post Info
	The Pub You can talk about anything here.	5,312	104,700	Jun 13 2014, 10:53 PM In: Character Change By: Walid
	Video Center Post videos that you have found. Porn = Ban.	1,047	3,106	Jun 6 2014, 03:39 PM In: So I've started doing t... By: Rubic
	Questions / Suggestions If you have any questions or suggestions post them here.	3,524	26,699	Jun 11 2014, 10:46 PM In: Avatar By: Alex
	Join New Members post your join forms here!	4,139	18,026	Jun 13 2014, 07:51 PM In: Joining as Akiko By: occupied!
	Roleplay School A place for those to discuss roleplaying techniques as well as seek help and mentoring from more experienced writers.	303	2,004	Jun 13 2014, 07:48 PM In: Maragai's School - Batt... By: Yakon
	Back To Alex's DBZ RPG	-	-	Redirected Hits: 64,317





• Members				
	Forum	Topics	Replies	Last Post Info
	Updates Post all updates in this board. Allowed 1 post per week. Post updates in the appropriate sub-board. Forum Led by: Monitors	18,739	51,058	Jun 13 2014, 11:20 PM In: Raine (Complete) By: Raine
	Character Vault Character biographies and histories.	164	135	Jun 8 2014, 02:55 PM In: Rubic By: Rubic
	Trophy/Customs Requests Forum Led by: Toma	815	4,672	Jun 12 2014, 05:06 AM In: Demonic Sprites By: Yakon
	Challenges & Recruiting Post challenges to other characters here or recruit people for your RPs. A challenge can be for a battle or even a spar. Forum Led by: Roleplay Guide	2,637	13,302	Jun 12 2014, 12:05 AM In: Revan By: Revan

Figure 1. An InvisionFree forum using a customized template

After you have the theme and style, it is time to start building the game technically. If you're going to use a website, then you will need to find a site host, create a layout, and finally start adding content to the site. If you're going to use a forum, then you will need to either write forum code or use one of the free online forum services such as *InvisionFree* or *Proboards*. Either one of these is a good choice because they both allow for customizable layouts that can be downloaded or you can even make your own. This adds more individuality to your forum which can set it apart from the thousands of other InvisionFree or Proboards forums out there. Building your own forum with phpBB allows for the most customization, however, and you can implement a lot more in-game features with phpBB boards such as character profiles with experience meters and word counters that automatically update a player's stats based on how many words they have written.

How "RPG-like" you decide to make your PbPRPG can vary considerably. Some game developers prefer creating complex systems for stuff like stats, combat, and leveling up. Other developers prefer making their PbPRPG much more centered on creative writing. Rather than having stats, you simply find other members of the site and write to them within a few standardized limitations such as "no god-modding," "no killing site NPCs," and "no auto-hitting." Be sure that you create a rules page that clearly lays out all in-game and out of character rules each player has to follow. This will help prevent disagreements among players, especially if they are battling which can get very complicated because neither side wants to lose.

When you are working on the RPG, it is a good idea to create everything live right on the site. Some webmasters would advocate working on your site offline and then publishing the whole thing in its entirety once it is completed. This is a mistake because it wastes valuable time. One of the advantages of working lively on your website is that you can continually test to see how things look after adding them. This makes it easier to tweak small details as you go along. Another advantage is that your site is "out there" longer for people and search engine crawlers to find. The longer your site has been around, the more credibility it will have in the eyes of both potential members and potential advertisers down the road. Plus, who knows, maybe someone will stumble across your site as you are working on it, think it looks cool, and bookmark it. There's a potential member right there!

Body			
Item	Cost	Build Time	Description
Karate Gui	500	1 week	Equipped on the body . Amplifies training by +8 speed. Can be worn with a leather suit or Capsule Corp Vest. If you level up a fighting style while wearing this item, you get the stat bonus of that style's level added to your stats as a permanent bonus.
Weighted Clothing	1,500	2 weeks	Equipped on the body . Amplifies training by +13 strength and +13 stamina. (+26 ki, +52 pl). If you level up while wearing this item, you get +1 bonus DP.
Super Weighted Clothing	3,000	4 weeks	Equipped on the body . Amplifies training by +22 strength and +22 stamina. (+44 ki, +88 pl).
Weighted Cape	1,000	2 weeks	Equipped on body . Amplifies training by +10 stamina and +10 strength (+20 ki, +40 pl). If worn with weighted Turban, will add an extra +2 strength and +3 stamina to training. If you level up while wearing this item, you get +1 bonus DP.
Capsule Corp Vest	1,000	2 weeks	Equipped on body . Adds +100 toughness in battle. Amplifies training by +15 vitality. (+15 ki, +30 pl). If equipped with capsule corp sword, amplifies training by +17 vitality instead.
Saiyan Armor W/ Shoulders	700	2 weeks	Equipped on body . Increases toughness in battle by +250 + 10% of your current toughness (does not increase ki or pl). Decreases damage dealt to you in battle by 8%.
Saiyan Armor W/O Shoulders	1,000	3 weeks	Equipped on body . Increases toughness in battle by +500 + 12% of your current toughness (does not increase ki or pl). Decreases damage dealt to you in battle 10%.
Saiyan Weights	500	1 week	Must be worn underneath Saiyan Armor. Amplifies training by +3 all stats (+12 ki, +24 pl)
Leather Suit	700	1 week	Equipped on body . Increases vitality in battle by +200 and amplifies training by +6 vitality.
Flame Tunic	800	2 weeks	Equipped on the body . Allows use of the technique flame shield when worn. Decreases damage of fire-based attacks done to you by half. Must wear inside Fire Temple.

Figure 2. A standard set of equipment for a Dragon Ball Z RPG

Just a few of the standard features that most RPGs have, are items, locations, and character stats. When it comes to items, make sure you add multiple pathways of items so that players can feel a sense of character customization through what they buy and equip. You should also consider the length of time it will take to purchase an item based on how easy it is to earn whatever money system you implemented. There should be at least three tiers of equipment: *starting equipment* that beginners can buy and use right away, *top-tier equipment* that people will ultimately save up for and is very strong, and then at least one tier in between these two that can serve as an intermediary. Once players start to attain the best stuff, you can release a patch that adds even better stuff. The more tiers of equipment you add, the better because it is always good to make sure that everybody has something to keep striving for. Item balance is something that you should spend a good amount of time on because it is one thing you need to get right from the get-go. Unlike other stuff which can be tweaked as the game goes on, changing items after people already have them can throw the whole RPG into disarray. It's not fair to "nerf" an item after players have already obtained it just because you think it is way too strong (though sometimes an item can be so game breaking you have no other choice). Imagine someone who spent months saving up for a god-tier item only to have that item nerfed after purchasing it. They will probably quit. That's one member lost because you didn't spend enough time on item balance.

The locations are the second standard feature among most RPGs. What type of locations you use will depend on what theme you settled on at the beginning of your development process. For example, if you are creating a fantasy RPG your universe will probably consist of cities spread across a continent. If you're making a science fiction, RPG on the other hand, your locations could be different planets spread across an entire galaxy. Locations should be diverse enough to encourage travel between them, but not too numerous so as to isolate the player base from each other. How many locations you add to the RPG will be dependent on how many players you have. A good rule of thumb is to create a universe where there will be a minimum of at least three active characters at each location at any given time. Why three? Because the average roleplaying

thread consists of about three people. Players in a PbPRPG want to interact with each others. The entire point behind the game is to roleplay with other people so make sure your universe isn't too vast.

The third standard feature is the player stats. Stats are typically categorized as either attack or defense. Some examples of attack stats include accuracy (for projectile weapons), strength (for melee weapons), and intelligence (for magic). Examples of defensive stats are toughness, vitality, stamina, and spirit (magic defense). Other commonly used stats are hit points, mana, speed, dexterity, charisma, determination, etc.

You can look to existing RPGs for some inspiration on stat categories that will fit with your RPG's theme.

When it comes to play-by-post RPGs, the focus should be on roleplaying so creating a combat system that incorporates player stats can be quite tricky. I would recommend avoiding a system that is purely math based since it will simply be too difficult to keep balanced in the long run. Formula based combat systems take months and months of testing to ensure nothing can be easily abused and that is just not what you want to be concentrating on when developing a PbPRPG. Not to mention that unless you can create a program to calculate damage, a math based combat system will scare off some potential players that don't want to sit there keeping track of numbers, calculating battle damage, and learning complicated formulas. Keep creative writing at the center of your battle system and instead use player stats merely as a guideline for what will be deemed acceptable versus what won't be. For example, it is believable that a character with a higher strength stat could throw his opponent through a table, but it wouldn't make sense for a character with a lower speed stat than his opponent to be dodging every attack. When you create your battle system, add battle referees to the mix. A battle referee is simply an impartial third member that vows to read every post about the battle and make rulings when the two participants have a disagreement or when he or she sees something they think violates the plausibility set forth by the statistical guidelines of each character.

Play-by-post RPGs are just as legitimate as big name video games and can often be much more fun. Play-by-post RPGs have been known to keep people occupied for years and have the added side effect of making people better writers in real life. Whatever other games do, PbPRPGs are not dead. They are still alive and well. If you're a fan of roleplaying games, give PbPRPGs a try by becoming a member or perhaps even creating one of your own.

User Actions Around MVW – Part 1

by Damian Czersnious

Composing a user interface is a quite challenging task due to the multitude interactions to handle. A user action involves one-to-many UI components such as buttons, text fields, etc., which makes things more complicated. For that reason, the MVW strategies exist. To a large extent, they focus on M, MV, C, P and V relationship devoting less attention to handling user actions. Thus, let's entirely focus on them and see how they fit the MVW mechanisms.

A group of *Model View Whatever* (MVW) design patterns shapes the relationship between data and its presentation. The business case of these structures is to deliver bespoke information (*that is the M*) to the custom-made view on a user request. From the engineering point of view, the *MVW* is thought-out and a proven structure that simplifies following the idea of *Separation of Concerns* (SoC) respectively to the application needs. The *MVC*, *MVP* and *MVVM* design patterns differentiate in understanding of the user interaction and data access.

All begins with the *Smalltalk MVC* (e.g. Described in a great dissertation of *Pattern-Oriented Software Architecture – A System of Patterns* page 125-143). The father of the later varieties, such as *Document-View* (used i.e. by *Microsoft*), *Visual Works MVC*, *MVC Model-1* and *MVC Model-2* gave birth to the *MVP* design pattern described by Mike Potel (Taligent IBM).

The original understanding of the *MVC*'s view and controller was reengineered when applied to the modern GUI frameworks in 90s and guys from *Visual Works* did a lot on that way. Therefore, the next generation pattern (*MVP*) replaces it with a greater focus on a user and an application relationship such as events and flexible model presentation. Without going into the details, this is a reason why the *MVP* is often seen as a generalised form of the *MVC*.

Finally, Martin Fowler focused on the consequences of the mechanisms shipped with the *MVC* and *MVP* in his *GUI Architectures* work. These days, designing a *UI* is about working with them. *Separated Presentation*, *Flow Synchronization*, *Observer Synchronization*, *Supervising Controller*, *Passive View*, *Application Model* and *Presentation Model* are the tools in developer's hand.

In my opinion, the *MV* of the *MVW* highlights a great achievement of the *Smalltalk MVC*, which is the domain and the presentation separation. Sometimes, the authors of other publications value separation of the business and the view logic most. Each of *MVW* patterns has own flaws and most of them concern exactly these two – *the business and the view logic*. However, constructions around domain evoke less emotions. Perhaps, engineers see them more universal as I do. The *whatever* (W) part stands for doesn't really matter rather than concrete C, P or VM.

The W originally stands for "whatever works for you", which used in the context of UI framework such as *AngularJS* implies that the framework is ready for all UI strategies and it's up to the coder which to use. However, I imply something different. From the solution point of view it doesn't really matter whether you go with *MVC*, *MVP* or *MVVM* (or any derivatives), because the mechanisms that work behind the seen are the true value of each. The trick is to understand them well and know which to use and when. So, mechanisms that crete them are far more important. It's also easier to understand them separately then a whole pattern at once.

The typical *UI* responds to many user actions. So, what is the type of thinking that would be helpful to get the job right? It turns out, that ensuring *source*, either by following *SOLID* principles or thinking about the effects that can be a good way to move forward. In general, the trick is to handle actions while staying compliant with the *MVW*.

Let's ponder the reasoning about the effects and *SOLID* principles. Let's try to formulate some rules for creating actions. As a code base, let's use the *Bakery* application – a training and a third-party free project: pure *Java* (*Mockito* for testing), *Vaadin* (without add-ons) plus *Hibernate*. In order to run it, please follow the following steps:

- Clone sources: git clone *<https://bitbucket.org/sanecoders/bakery.git>*
- Navigate to bakery folder
- Execute maven command: mvn package jetty:run
- Navigate to: *<http://localhost:8080>*

Handle one Task Within an Action Method

This is a straightforward translation of the Single Responsibility Principle and OpenClosed Principle adjusted to the action's nature.

Scenario 1

The bakery application uses a product manager that doesn't only have a product save action, but a save and back action as well, which redirects user to the previous page. This is because, the bakery owner expects to list products and modify them on demand. So, a pretty natural expectation is to navigate back after finishing editing and see the updated product list. From the user perspective, saving and returning back is one action. Usually, this is an unspoken expectation that we – engineers – should be aware of. It'd be against human nature asking users to navigate manually back after the invoked save action. The fundamental utilization of OOP (*Object Oriented Programming*) paradigm is to reflect the existing relationship of the world. The bakery owner, while working with the list of the bakery products, treats save and back editor's action as a single click; as finishing editing – we say “Yes, of course, save my changes and let's go back to the list”. This is the use case to design.

At the beginning we need to model proper application context by defining right package. Then, create action class: `com.sanecoders.bakery.product.edit.view.ProductSaveAndBackAction`.

The SRP won't be violated since only one actor will be interested in this action. Sometimes, it helps to replace in mind “SaveAndBack” with e.g. “EditorFinish” to hear `ProductEditorFinishAction`. However, this is just an exercise to get the proper understanding of the situation. Such name would be less descriptive and since we already understood the context, it's expected to move on and learn what the action is about?

Keynote

It's a good practice to begin studying classes by reading their package names. A package gives the functional context in which interesting class operates. For that reason, it's so essential to name them well.

Scenario 2

There is another unspoken user expectation that we need to address correctly. Every functionality we build has to be extendible, since reusing working solutions already lies in the human nature. Thus, one day, bakery manager may ask for editing products in a pop-up window, which simply makes redirecting back too much. It'd be quite unfair to spend time on modifying something we had to accomplish in steps earlier. Each step is a task to be done. One is to save product (`ProductSaveAction`) while the second is to navigate back (`ProductSaveAndBackAction`). The second is a simple extension of the first one. This is how a user sees this and so should we. We may have two implementations.

First, Mike Potel doesn't say how to handle the view logic from the action. He just focuses on the business part.

```
public class ProductSaveAction implements Button.ClickListener
{
    public ProductSaveAction( ProductEditPresenter productEditPresenter ) { ... }

    @Override
```

```
public void buttonClick( Button.ClickEvent event )
{
    productEditPresenter.save(); // Potel's delegation
}

public class ProductSaveAndBackAction extends ProductSaveAction
{
    @Override
    public void buttonClick( Button.ClickEvent event )
    {
        super.buttonClick( event );

        // Potel doesn't say where/how to handle UI code
        event.getButton().getUI().getPage().getJavaScript().execute( JAVASCRIPT_GO_BACK );
    }
}
```

Second, however Supervising Controller (the mechanism named by Martin Fowler but described by Andy Bower and Blair McGlashan), version does care about the view logic (ProductEditPresenter).

```
public class ProductSaveAndBackAction implements Button.ClickListener
{
    @Override
    public void buttonClick( Button.ClickEvent event )
    {
        productEditPresenter.saveAndGoBack(); // Bower and McGlashan's delegation
    }
}

public class ProductEditPresenter
{
    public void save() { ... }

    public void saveAndGoBack()
    {
        save();
        productEditDisplay.goBack(); // controller supervises view
    }
}
```

The trick is to understand that we have one user action and two tasks (two software actions) that come from spoken and unspoken user expectations. Depending on the chosen strategy, we may see them as two action classes or two controller methods.

But, sometimes we're in a hurry and for much more complex situations, we may have no time to brainstorm. This is why SOLID principles are so helpful. The OpenClosed Principle requires us to write code that is open for extension, but closed for modification. Saving and navigating back can't be done within only one method, because according to the rule, it's forbidden to modify the saving functionality with unrelated code.

Keynote

People build things (e.g. Buildings) by extension (e.g. They build second floor on the top of the first one). Building a software is not any different. This is why a software has to be always expansible. This is an unspoken client expectation – one of the most anticipated, yet often violated and unaccepted when exposed to the customer.

Scenario 3

After finishing the editing, the product manager navigates back, thereby happy bakery owner may see an updated product list. The expectation is to update the list during navigation. We know already, it has to be done in steps. Updating product list should take place before or after the navigation. Switching pages is a task itself and typical web *UI* framework will do it for us.

```
public class GotoProductListAction implements ViewChangeListener
{
    @Override
    public boolean beforeViewChange( ViewChangeEvent viewChangeEvent )
    {
        return true;
    }

    @Override
    public void afterViewChange( ViewChangeEvent viewChangeEvent )
    {
        if( ProductView.LIST.getName().equals( viewChangeEvent.getViewName() ) )
        {
            productListPresenter.show();
        }
    }
}
```

In the future, we may use a – before method – e.g. to grant access to the product manager only of the bakery staff. Granting access and refreshing page won't violate the SRP since there is one actor (bakery owner) who is interested in the product manager and zero actors interested in its partial use. On the other hand, it would violate the principle, if both, customer and bakery staff can access the product list, but not to edit. Then, we have two actions. One is for refreshing and the other is for granting access e.g. *GotoProductListAction* and

AuthorisationController.

Depending on the chosen technology of authorization, the implementation may differ e.g. Java Web and EJB container ship with internal security mechanism. This container managed solution that won't fit all Java web *UI* frameworks such as *Vaadin* views – here *App Foundation* plug-in might help.

This is a true user friendly rule that focuses entirely on the action itself. To conclude with one task per an action method, we have to think like a user and catch his spoken and unspoken expectations. We also have realized that often small steps can be grouped and seen as a whole (product saving and going back, navigating to and refreshing the product list). Nevertheless, these steps are important from the *SoC* point of view and SO of the SOLID tools that draw our attention to them.

Follow Encapsulation

The best bakery owner tracks sells to determine the optimal supply of flour among other things. He holds a pen and a piece of paper on the counter to easily take notes of each sale. One day, bakery owner asked for a software that would do the job for him, so he can spend more time with his family.

The prediction of the flour size represents the human need. Tracking sells and doing calculations are middle activities. Therefore, the requested functionality can be broken down into two modules. *First*, tracking sales and the *second*, for calculations. Let's go with the first one.

In order to take a single note of sale, a human being has to mark a sold product on the product list. At the beginning, the human brain asks the body to move a hand. That is a gesture. Finally, a pen passes ink on the paper drawing some shapes. The software business logic represents the human need, which is to remember, *the sale*. The gesture is a software action. The view logic represents the shape of the mark. The hand is a keyboard or a mouse button and the pen is a display.

This is a good example of the *Supervising Controller*. The controller (the human brain) administrates the realization, of both the business and the view logic.

```
public class MarkProductAction implements HandHoldingPen.HandTickGesture
{
    private final ShopAssistant shopAssistant;

    public MarkProductAction( ShopAssistant shopAssistant )
    {
        this.shopAssistant = shopAssistant;
    }

    public void tickGesture( HandHoldingPen.Tick tick )
    {
        shopAssistant.markProduct( tick.getId() );
    }
}

public class ShopAssistant
{
    private final SaleMemory saleMemory;
    private final Pen pen;

    public ShopAssistant( SaleMemory saleMemory, Pen pen )
    {
        this.saleMemory = saleMemory;
        this.pen = pen;
    }

    public void markProduct( ProductId id )
    {
        pen.drawMark( id );
        saleMemory.rememberSale( id );
    }
}
```

From the previous examples, we know that sometimes the view logic may stay apart from the controller's care. This may happen e.g. when the bakery owner uses the cash register. The difference is that human being doesn't have to know how to draw a mark anymore. The cash register knows that for him.

```
public class MarkProductAction implements HandHoldingCashRegister.HandTouchGesture
{
    public MarkProductAction( ShopAssistant shopAssistant, CashRegister cashRegister ) { ... }

    public void touchGesture( HandHoldingCashRegister.Touch touch )
    {
        cashRegister.printSale( touch.getId() ) // delegates view logic
        shopAssistant.rememberSales();          // delegates business logic
    }
}

public class ShopAssistant
{
    public ShopAssistant( SaleMemory saleMemory, CashRegister cashRegister ) { ... }

    public void rememberSales()
    {
        saleMemory.updateSales( cashRegister.getPrintedSaleIds() );
    }
}
```

```
}  
  
public class CashRegister  
{  
    public CashRegister( Pen pen ) { ... }  
  
    public void printSale( ProductId id )  
    {  
        pen.drawMark( id );  
    }  
}
```

In both versions we expect actions neither to do the business, nor the view work. However, we do expect them to connect triggering component – a hand or some button from a common *UI* – with the business and the view logic.

A human being knows how to hold a pen in a hand. This explains, why actions are *UI* framework oriented. The human brain will oversee the hand holding the pen in a different way, then the hand holding the cash register because of their assorted natures. In other words, gestures (actions) depend on their subjects.

Let's go back to the Bakery application, The `ProductSaveAndBackAction` follows the *Supervising Controller* strategy (`HandHoldingPen` equivalent) while `ProductsStartStopEditingAction` takes Mike Potel's way (`HandHoldingCashRegister` equivalent).

Additionally, the part of the navigation (going back) is quite an interesting one. It belongs neither to the product edit view logic nor the product save and go back action logic. However, it does belong to the *UI* framework (to the human body – eyes). This is how I explain myself, why action may hold a reference to its caller. After all, both, edit view and its actions are *UI* framework oriented, both can be seen as one presentation component. Based on that, either the view or action may handle navigation, *if Potel's solution preferred*.

```
public class MarkProductAction implements HandHoldingCashRegister.HandTouchGesture  
{  
    public void touchGesture( HandHoldingCashRegister.Touch touch )  
    {  
        touch.getHand().getHuman().getEyes().goBackToClient();  
    }  
}
```

With this occasion, it's worth to mention the Passive View, where the whole view logic is done by a business unit (Controller or Presenter). The difference with the Supervising Controller is that such controller just administrates the view logic.

Now, how this all refers to the encapsulation? Well, the shop assistant's brain is both the business and the view logic unit (logic, not entity). It knows how to store single sale in the brain's memory and draw a mark. A source code represents that truth. The business unit associates with storage (brain's memory – entity) and the view (pen – entity). The business and the view units (objects) are responsible for processing its algorithms (brain's way of thinking – logic). We may change the way of thinking only through the brain and that is the example of the encapsulation. Thereby, we encapsulate all necessary internals within classes and change them through its API. If we see action as a gesture, then following encapsulation means to call *API* of the right logic unit that encapsulates the right algorithm (but not to do the job by itself).

Treat Action as a Customization

This rule additionally underlines the unspoken user expectation that we have described earlier. A software has to be always extensible.

One day, the bakery owner may change his mind and ask to edit the product in the pop-up window rather than on a separate page. The natural way of providing requested change is to reuse editor module attached to the new save and close pop-up action. If previously designed save and navigate back action works as an extension, so it's only about replacing action in the *Main Pattern* (e.g. In editor's builder `ProductManagerBuilder`).

```
public class ProductManagerBuilder
{
    void buildProductEditor()
    {
        buildProductEdit();
        buildProductSaveAction();
    }

    void buildProductSaveAction()
    {
        LangButton saveComponent = productEditViewFactory.createSaveButton();
        saveComponent.get().addClickListener(
            productEditViewFactory.
createProductSaveAndBackAction( productEditPresenter ) );
        productEditView.setSaveComponent( saveComponent );
    }
}
```

Otherwise, the modification of the editor's source code will be necessary, which will disappoint the customer and violate *Open Closed Principle*.

Keynote

A software is extensible when properly mirrors world's relationships. The SOLID principles are technical translations and generalizations of the observed patterns. Among other tools, they help us to mirror these complexities without thinking about them.

Summary

Software actions physically represent user gestures. Their responsibility is to connect the user intention with implementation. Such understanding results not only with the better functional encapsulation, but also with increased readability. The readability of the source code is the most decisive engineering factor of the project success.

Separating business, view and action logic helps to adjust (re-factor) source code structures respectively to the current needs. It's quicker to invert dependencies and move functionalities between modules to get a modular solution where all parts are separately deployable. It's also easier due to the automatic nature of the work. The complexity of the structures or chosen strategy (e.g. Described by Martin Fowler) depends upon functional and non functional requirements (not covered in this article). In order to play with these structures, the source code has to be written in a modular and self explanatory way and it needs to follow the *SOLID* principles. This will also be the main subject of part 2 – “associations” of this article.

On the Web

GUI Architecture by Martin Fowler <http://martinfowler.com/eaDev/uiArchs.html>

About the Author

Damian Czernous is a software engineering coach at Nokia Networks, who wants to share his knowledge with others. He is passionate about reasoning in engineering. This is an example of his work. Feel free to comment on it!

User Actions Around MVW – Part 2

Associations

by **Damian Czernous**

From my observations, the way we handle user actions influences the modularity of the UI most, Why? In a single MVW module, we usually have several user actions that tightly or loosely couple code. The view sets UI components and may handle UI logic. The UI business unit handles the UI business logic. But, it's the user action that holds the key to both "worlds". Multiplying that fact by the number of actions in a single module gives a clue about the potential mess. What's more, action's execution flow is more challenging to track, because it has tendency to be transparent; usually it's triggered by UI and occasionally by our code. Thus, the way we control action associations controls "the mess".

Last time, we spoke about handling user actions in MVW oriented applications ("User actions around MVW, part 1"). We have formulated simple rules based on spoken and unspoken user expectations and we've tried them out by the following several strategies known from the MVW. In a result, we identified what are action's, business's, and view's logic obligation. Finally, we've proved the discussed concepts with a source code. This time, we're going to think about action associations rather than internal organization, think about the consequences of the communication. As before, we'll use Bakery application (*Java* with *Mockito*, *Vaadin* plus *Hibernate*) to exercise our thinking. In order to run it, follow the steps:

- clone sources: `git clone https://bitbucket.org/sanecoders/bakery.git`
- navigate to bakery folder
- execute maven command: `mvn package jetty:run`
- navigate to: `http://localhost:8080`

The Bakery application consists of several functionalities. A product manager allows to list, add and edit products. User management features support signing in, out, and switching between available languages. All these functionalities can be deployed independently and the decision about their usage comes from the *Main Pattern*. In other words, it's possible to compose another Bakery application, e.g. Bakery Quick View that will provide a product overview (no modify option) displayed in a few languages (no authentication and authorization) just by delivering another *Main Pattern*.

This is a good example of the modular application that allows reusing existing functionalities (modules) without modifying them. Modules that are ready for re-use, satisfy the unspoken client expectation, which is "Of course, once completed, I would like to use my features freely". For the web applications, handling user actions has a significant impact on such result. Often, they communicate through available functionalities, e.g. by letting the editor (module) to display or add a product form from the list (module).

In practice, action implementation may simply delegate display request to the editor module e.g.

```
package com.sanecoders.bakery.product.list.ui;
public class ProductAddAction ...
{
    private final ProductEditorPresenter productEditorPresenter;
    public ProductAddAction( ProductEditorPresenter productEditor )
    {
        this.productEditorPresenter = productEditorPresenter;
    }
    @Override
```



```
public void buttonClick( Button.ClickEvent event )
{
    productEditorPresenter.onAdd();
    getNavigator( event ).navigateTo( "productEditorPage" );
}
}
package com.sanecoders.bakery.product.edit;
public class ProductEditPresenter
{
    public void onAdd()
    {
        showEmpty();
    }
}
```

The method `buttonClick()` works in the context of the `ProductAddAction` class, thus the method name, from the functional point of view can be understood as `onAddProductRequest()`. Using UI frameworks, we implement generalized interfaces, therefore the only place where we learn about that context is the class name. This should be a good motivator for proper naming and maintaining class names for the time being.

On the add product request, proposed action asks editor's presenter to prepare editor's view by filling a form with empty data. Finally, it redirects to the `productEditorPage` making the prepared form to be visible to the user.

A drawback of this solution is the tight coupling of the list and edit modules. The action holds a direct reference to the edit module (presenter class) being a part of the list module (notice packages). A better solution would maintain modules independency (between `ProductAddAction` and `ProductEditPresenter`) and keep, at the same time, the existing flow of control.

```
package com.sanecoders.bakery.product.list.view;
public class ProductAddAction ...
{
    @Override
    public void buttonClick( Button.ClickEvent event )
    {
        productEditorInteractor.onAdd();
        getNavigator( event ).navigateTo( ProductManagerPages.EDIT.getName() );
    }
}
```

```
package com.sanecoders.bakery.product;
public interface ProductEditorInteractor
{
    void onAdd();
}
package com.sanecoders.bakery.product.ui;
public enum ProductManagerPages
{
    EDIT( "product.edit.productEditPage" );
}
package com.sanecoders.bakery.product.edit;
public class ProductEditPresenter implements ProductEditorInteractor
{
    @Override
    public void onAdd() { ... }
}
```

After changes, both the list and edit modules, in order to communicate, need to satisfy the contract of the product manager (`ProductEditorInteractor` and `ProductManagerPages`).

Application modules are little pieces that become meaningful when used within some context. In this case, little modules compose a product manager module that composes, along with the user module, the Bakery application. In other words, product list end edit need to satisfy the contract of the platform they are part of. For that reason, above the interface and Enum class reside in product and “product. ui” packages and are called the *API* of the product manager. This is a frequent situation when we write a client application for a back-end system, e.g. in order to run a game on Facebook, the game itself needs to satisfy Facebook *API*.

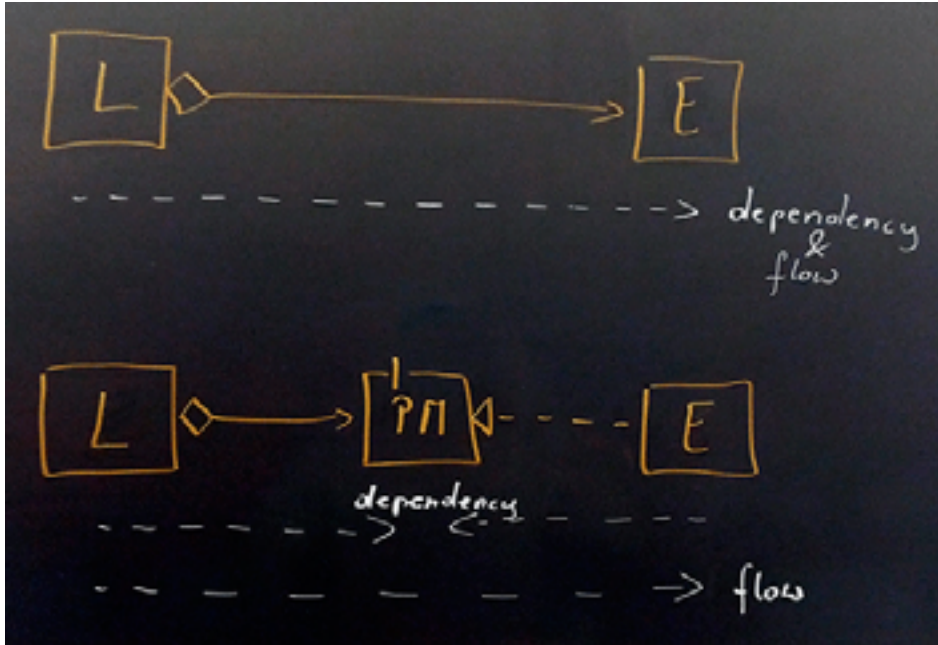


Figure 1. Inverting dependencies between List and Edit with Product Manager

Keynote

The package, class and public method names build sentences from which we can learn the application. For the engineer, having the naming skills on the right level helps to create self-describing source code and simplifies following *DDD* practice. By introducing an interface, we invert dependencies and keep the original flow of control. Combining naming and inverting dependency practice, we're in a far better position to design a well organized higher level *API*. Application features represent spoken user expectations. They're the reason why we build applications. While coding, we (engineers) also need to remember about the unspoken expectations such as reusability and extensibility. That means, asking questions for the less obvious situations.

For the client or company we work for, it might be important to provide several user interfaces or be able to replace current one. In such cases, *UI* related part should live independently. Therefore, every *UI* oriented module may consist of *UI* business and *UI* view modules. With technique described above, we can easily approach this need. The problem happens, when we ask questions and our client isn't sure or after a while he changes his mind. The funny thing is, that the variability of the human nature is the inherent part of this world, thus we need to take care of it as well. For that reason, I prefer saying, that the application has to be always ready for extension. I am strongly convinced that we should ensure extensibility of the entire solution. The Bakery application is extensible, but currently it doesn't support separate deployment of the *UI* part within each module. However, it's easy to make that change by extracting a few interfaces (no algorithm changes or longer refactoring is required). It's actually very automatic work and most *IDEs* have supporting refactoring tools.

```
package com.sanecoders.bakery.product.edit.ui;
public class ProductEditView
{
    private final ProductFieldGroup productFieldGroup;
    public ProductEditView( ProductFieldGroup productFieldGroup )
    {
```

```
        this.productFieldGroup = productFieldGroup;
    }
}
package com.sanecoders.bakery.product.edit.ui
public class ProductSaveAndBackAction
{
    private final ProductEditPresenter productEditPresenter;
    public ProductSaveAndBackAction( ProductEditPresenter productEditPresenter )
    {
        this.productEditPresenter = productEditPresenter;
    }
}
package com.sanecoders.bakery.product.edit;
public class ProductEditPresenter
{
    private final ProductEditDisplay productEditDisplay;
    private final ProductService productService;
    public ProductEditPresenter( ... )
    {
        this.productService = productService;
        this.productEditDisplay = productEditDisplay;
    }
}
```

The edit module follows Andy Bower and Blair McGlashan *MVP* design pattern. The view (`ProductEditView`) has no direct reference to the business logic that is represented by `ProductEditPresenter` class. All view and business oriented components are separated by package (edit and edit.ui). This makes eventual UI module creation simpler. Only action (`ProductSaveAndBackAction`) holds the direct reference to the presenter, but as mentioned before, it's just about extracting the presenter interface. This interface will be part of the module's *API*.

```
package com.sanecoders.bakery.product.edit.ui
public class ProductSaveAndBackAction
{
    private final ProductEditPresenter productEditPresenter;
    ...
}

package com.sanecoders.bakery.product.edit;
public interface ProductEditPresenter { ... }
public class ProductEditPresenterImpl implements ProductEditPresenter { ... }
```

When a solution needs to support exchangeability of the *UI* part, it sounds reasonable to use the *Supervising Controller* mechanism (described by Martin Fowler in GUI Architectures work), which makes *UI* implementation more intuitive (find examples in “Follow encapsulation” chapter of the previous article for reference).

Action as a Part of Another Class

A typical situation happens when we expect a view or a business class (depending on the chosen strategy) to be small enough and to support one action of a kind. Have a look on the login view class.

```
public class LoginView implements Button.ClickListener
{
    private LangTextField emailField;
    private LangPasswordField passwordField;
    private LangButton signInButton;
    @Override
    public void buttonClick( Button.ClickEvent event )
    {
        try
```

```
{
    signIn( event );
}
catch( LoginModel.UserValidationException e )
{
    notifyFailedAttemptToSignIn( event );
}
}

private void signIn( Button.ClickEvent event ) throws LoginModel.UserValidationException
{
    loginModel.validateUser();
    event.getButton().getUI().getSession().setAttribute( BakerySession.USER.name(),
loginModel.getUserId() );
    event.getButton().getUI().getPage().getJavaScript().execute( JAVASCRIPT_GO_BACK );
}
...
}
```

The `LoginView` class is considered the sign in the form and the sign in operation. The key thing is to be aware of that, and implement these aspects independently; like two separate classes combined into the one – they share common space, but not states. Therefore, `buttonClick()` method should not rely on the view states, but only on those that apply to the action itself.

In our example, the `signIn()` method doesn't use class `signInButton` field directly, but accesses it by making a call to `Button.ClickEvent.getButton()` method using action's API. This is because, the `signInButton` field composes view's login form – belongs to the view class.

From the architectural point of view, such practice allows to extract the action code to the separate class when needed. This kind of thinking guarantees loose coupling between roles, even if they're coded within the same class.

For many, this kind of thinking is the reason for such code simplification. For me, it's a nice side effect of the thinking about responsibilities under favorable circumstances. When I expect a class to be very simple (favorable circumstances), I can merge implementations that support a wish of the same application actor, e.g. in order to sign into the Bakery application, the user wants both access sign in form and to proceed with the sign in operation. This is a good example of the design shortcut that we may benefit from. For such practice, it's essential to have application actors neither interested in the view nor the action independently, since that would violate the *Single Responsibility Principle* (SRP).

Summary

The Mike Potel or Andy Bower and Blair McGlashan way of handling actions are two available options to choose from. To gain deployment independence between application functionalities or `UI` code, we may use the *Invert Dependency* technique with each. Actions respond to the user gestures captured by the view components such as buttons. Actions are also `UI` framework oriented. It makes a lot of sense to keep them close to the `UI` parts such as views. This makes the code organization more readable and intuitive.

Action associations should point towards business and (or) view code units, but not vice versa. The best place to connect actions to their triggering components is the *Main Pattern*.

About the Author

Damian Czernous is a software engineering coach at Nokia Networks, who wants to share his knowledge with others. He is passionate about reasoning in engineering. This is an example of his work. Feel free to comment on it!

CMS-Based Web Application Maintenance Made Easy with Integrated OO Design

by Jean-Pierre Norguet

Build your web application upon a CMS and use OO to get 3 advantages:

- 1. Speed: immediate prototyping and development support*
- 2. Reliability: fast, readable, and consistent bug fixing*
- 3. Adaptability: flexible change request implementation*

Nowadays Content Management Systems (CMS) with rich set of features allow fast prototyping, extended reuse, and accelerated development of full-featured web applications. However, each CMS comes with its own framework, library, and database schema, therefore limiting code readability and portability. Integrating your web application with a CMS can clutter your code readability with non-trivial CMS-specific statements, which makes maintenance difficult. With the approach that we propose, layered OO design and persistence integration with the CMS and the database make the code simple and readable. In this article, we study a simple design and implementation of this approach, from high-level overview to detailed pieces of code. The code implements a typical “course session display” web application and is run in Wordpress, a popular rich-featured CMS based on PHP and MySQL.

Most CMS like Wordpress, Drupal, or SPIP use an architecture based on three parts (Figure 1):

1. A framework that organizes user calls into structured requests;
2. A set of standard screens that handle typical document display and is extensible to any kind of business request management;
3. An extensive programming library to manipulate the CMS database elements.

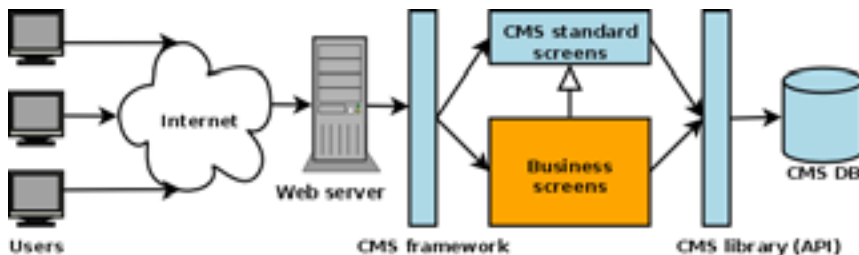


Figure 1. Typical CMS Architecture (CMS code in blue, custom code in orange)

From left to right, the users of the web application – or the visitors of the web site or blog – send HTTP requests from their browsers through the Internet to the host web server. The web server is configured to pass the request to the CMS framework, which structures the request and passes it to the presentation screens, whether they are standard screens displaying the CMS information content or custom business screens doing something useful. Screens are typically mixed PHP-and-HTML pages that query the CMS database, retrieve the content, and display it the desired way. The CMS database is formatted to store CMS-specific content like posts/pages, users, comments, and various other data that can be managed by third-party extensions – most of the time named “plugins”.

This is true for most web-enabled CMS that can be found today on the Internet. For the simplicity of the reading, we will focus on the Wordpress CMS. In Wordpress, the middle part that handles document display and allows for business extension is called *theme*. The Internet is full of free and non-free Wordpress themes that offer extremely various ways of displaying the same information. Prototyping a website or blog based on theme selection is extremely fast. Also, separating presentation (in the theme) from content (in the CMS database) allows website look-and-feel change at any time without any modification. If you are unfamiliar with Wordpress, the relevant elements of the Wordpress framework, API, and database schema will be discussed in time in the following.

The typical process for website prototyping based on Wordpress comprises the following steps:

1. domain name reservation through a registrar, or reuse of a subdomain
2. web host reservation supporting PHP and MySQL
3. Internet download of Wordpress files and upload to host
4. creating a Wordpress-dedicated MySQL database
5. configuration and installation of Wordpress, which creates the database schema with some basic content
6. choice of the standard theme that comes with Wordpress or an existing theme downloaded from the Internet
7. configuring the display and create or import some content
8. tailor and extend the Wordpress theme according to business needs

In the following, we assume that you are familiar – or can become so, as Wordpress installation is extremely easy – with the first steps of the process. We focus on the last point: extending Wordpress to the business needs. We assume these needs cannot be met or satisfyingly met with the use of plugins, especially when application complexity is growing over time. Striving for simplicity, we leave out the addition of powerful and complex frameworks like Zend or Symfony, although large projects may benefit of their use.

Sample Example

Let us consider a simple example that will illustrate our approach. We are in charge of a national art academy in which teachers located all over the country give similar courses. We need to display the sessions per course or per teacher so the students can choose the sessions that suit them best: close to their place and at available time. Teachers need a web interface to encode their course sessions. Administrative staff encodes the course descriptions and manage access to the application.

Example Features: Use Cases

A basic use case diagram for this applications is as follows (Figure 2):

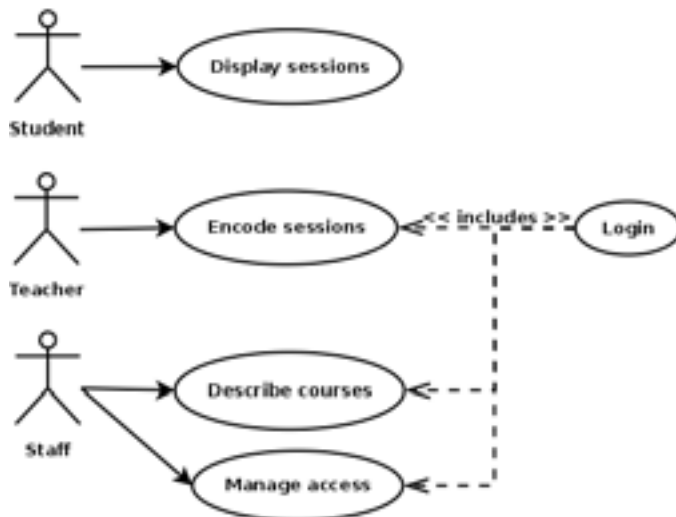


Figure 2. Use case diagram of our sample application

The diagram shows the three kinds of actors: student, teacher, and staff. Teachers need a login to realize their session-encoding use cases. So do the administrative staff members, with a different clearance level, to encode the course sessions and manage access to the application, that is providing login and passwords to the teachers and incoming staff members. The academy being open to the general public, students need no login to search, display, and browse course sessions. The students need for a login might change if the system would implement an online session reservation system, but let us keep it simple for the example.

System Model: Object Classes

In the direct line of standard OO development processes, let us model the system objects. The UML class diagram is designed as follows (Figure 3):

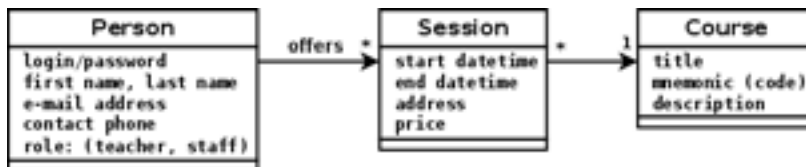


Figure 3. UML class diagram of our sample application

- The *Person* (or User) class models the actors proxies: teachers and staff members, with an attribute that defines their role. Students might be modeled as well if the application would keep the record of each student for reference. To keep it simple, we assume the student database is located elsewhere, for example in standalone Excel files. Teachers and staff members have a login and password to access the application. Their name is of course stored, as well as an e-mail address and a phone number to contact them.
- The *Session* class models the heart of the system information: the available sessions that teachers offer to the students. We assume the teachers give only courses defined in an existing set the academy has chosen to offer. Each session starts and ends at some date and time, the teachers need to specify. The address is also chosen by the teacher, as well as the price, which we assume to be included in a reasonable range arranged between the academy and the teacher.
- Finally, the *Course* class models the available courses in the academy. Teachers willing to work with the academy must provide sessions for the defined set of courses, with the necessary skills and diplomas. Course descriptions are set by the academy board, and encoded by the administrative staff. Teachers cannot change them, although they can give their courses the way they want. Each course has a title, and a mnemonic code for reference.

Database Mapping

In a web application that would not be CMS-based, the class diagram would map to corresponding database tables, to be created. As we integrate with the Wordpress CMS, we can reuse CMS database tables and features, including login/password security management and visual course description editor. In our sample application, the class diagram could be mapped to the CMS database schema like this (Figure 4):

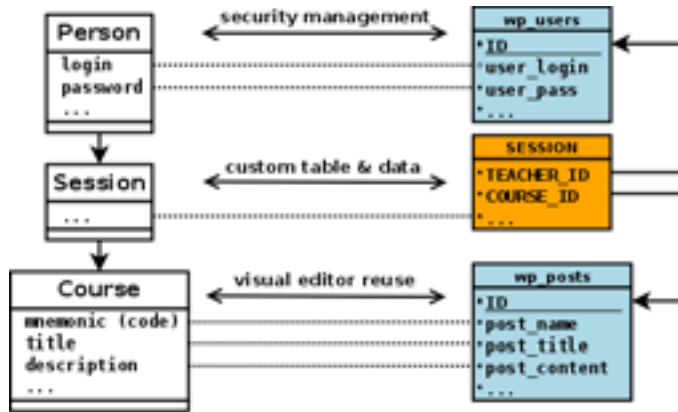


Figure 4. UML to DB mapping (reused CMS elements in blue)

The following Wordpress elements – and the corresponding features – can be reused as follows (note that the default Wordpress table names start with the `wp_` prefix):

- *Security management*: the login and password fields of the Person class can be mapped to the `user_login` and `user_pass` fields of the `wp_users` Wordpress table. Reusing these two fields gives more than just some table and field reuse: the Wordpress login feature uses them and the security access becomes automated by the Wordpress system, without requiring any custom security code to be written.
- *Custom code*: the Session class is a newsystem element that the CMS does not model as is. We need to create the database table and all the needed fields. However, we can link secondary keys to the primary keys (ID fields) of the `wp_users` and `wp_posts` tables, which seals the integration of the custom code.
- *Visual editor*: the mnemonic code, title, and description of the course can be mapped to the `post_name`, `post_title`, and `post_content` fields of the `wp_posts` Wordpress table. This allows the update of these fields by the built-in Wordpress visual editor, with all the word-processing-like composition features needed by the non-technical administrative staff (Figure 5).

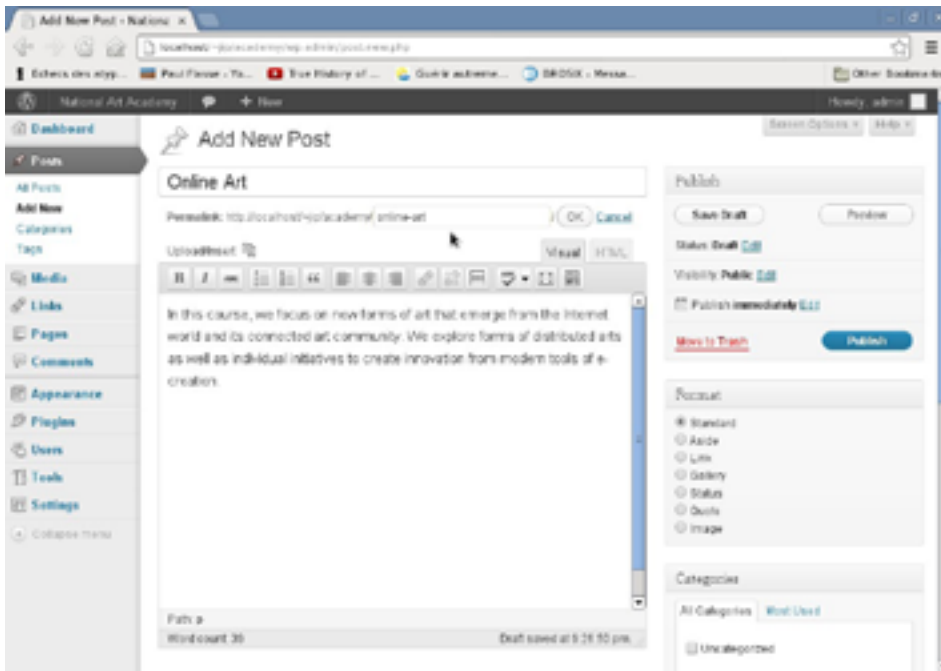


Figure 5. Encoding course description in Wordpress

This interface is convenient in many ways. First of all, it needs no custom code to be written: the administrative staff connects to the Wordpress system and receives immediate access to the post management menu (see it expanded on the left). In our mapping, we have defined that each post corresponds to a course given in the academy. Post editing becomes a handy way to course editing for the staff. The visual editor interface shows a field to write the course title. Right under it appears the permalink, with a small field to complete the `post_name`, which corresponds to the course mnemonic code. Finally, the text area allows the staff to edit the course description, with the composition toolbar, image insertion facility, as well as HTML edition panel for the technical savvy.

As an exercise here, you can open the Wordpress interface and create a few posts describing academy courses, along with some user access for a teacher and an administrative staff member. Another exercise could be to create the custom Session database table and populate it with two or three rows. Here are some table creation and row insertion statements:

```
CREATE TABLE SESSION (ID BIGINT(20), TEACHER_ID BIGINT(20), COURSE_ID BIGINT(20),
    START DATETIME, END DATETIME, ADDRESS VARCHAR(255), PRICE INT);
INSERT INTO SESSION VALUES (1, 3, 5, '2015-03-22 09:30:00', '2015-03-23 18:30:00',
    'Madison Cultural Center, Main Square, 45032 Rogerville', 160);
INSERT INTO SESSION VALUES (2, 4, 5, '2015-04-04 08:45:00', '2015-04-05 19:15:00',
    'City Hall Amphitorium, Central Station Avenue 1A, 93012 Sallytown', 200);
INSERT INTO SESSION VALUES (3, 4, 14, '2015-05-02 08:45:00', '2015-05-04 19:15:00',
    'City Hall Amphitorium, Central Station Avenue 1A, 93012 Sallytown', 300);
INSERT INTO SESSION VALUES (3, 4, 16, '2015-05-21 08:45:00', '2015-05-21 19:15:00',
    'City Hall Amphitorium, Central Station Avenue 1A, 93012 Sallytown', 120);
```

In the insert statements, make sure the `TEACHER_ID` and `COURSE_ID` correspond to actual IDs in your Wordpress database.

Extending the CMS Tables

As we have seen and will implement later, the security management and visual editor features can be reused from Wordpress by mapping to the `wp_users` and `wp_posts` tables. However, these tables do not contain the necessary fields to model the information defined in the system classes. For example, the `wp_users` table contains a `user_email` field for the e-mail address but does not contain a field for the contact phone. Two

approaches compete here: create a new custom table with the needed complementary fields or use the wp_usermeta table convenience offered in Wordpress.

- The new custom table option is more flexible: it is generic and allows the creation of any number of fields.
- The Wordpress wp_usermeta table works only for extending the wp_users table, but no table creation operation is needed.

In addition, we need fields for the first name and last name, as well as for the role the user plays in the application (teacher or staff member). The first and last names are already implemented by default in Wordpress and are located in the wp_usermeta table. The data can be easily accessed using the Wordpress API. The role can be mapped to the Wordpress security clearance level system. For example, we may define that teachers get the subscriber level, which is the lowest level, and that staff members get the editor level, which is needed to encode courses. Check the Wordpress documentation to view the level privileges in details. The other option is to declare a custom table field, which has the advantage of being CMS independent.

Here is the PHP code that supports teacher retrieval and storage from the Wordpress schema:

```
public function getByLogin($login) {
    $person = new Person();
    $wp_user = get_user_by('login', $login);
    $person->id = $wp_user->ID;
    $person->login = $login;
    $person->first_name = get_user_meta($wp_user->ID, 'first_name', true);
    $person->last_name = get_user_meta($wp_user->ID, 'last_name', true);
    $person->email_address = $wp_user->user_email;
    $person->contact_phone = get_user_meta($wp_user->ID, 'contact_phone', true);
    return $person;
}

public function update($person) {
    $wp_user = get_user_by('login', $person->login);
    update_user_meta($wp_user->ID, 'first_name', $person->first_name);
    update_user_meta($wp_user->ID, 'last_name', $person->last_name);
    $wp_user->user_email = $person->email_address;
    update_user_meta($wp_user->ID, 'contact_phone', $person->contact_phone);
}
```

We leave as an exercise the writing of the code for retrieval and storage of the complementary fields from a custom table. It is a mix of some of the above Wordpress API calls and additional SQL statements. The SQL code can be inspired from the retrieval and storage of sessions:

```
protected function rowToSession($row) {
    $session = new Session($row->ID, $row->TEACHER_ID, $row->COURSE_ID);
    $session->start = $row->START;
    $session->end = $row->END;
    $session->address = $row->ADDRESS;
    $session->price = $row->PRICE;
    return $session;
}

function getSessions() {
    $query = "SELECT * FROM SESSION ORDER BY START";
    $stmt = Connection::getInstance()->prepare($query);
    $stmt->execute();
    $sessions = array();
    while ($row = $stmt->fetchObject()) {
        $sessions[] = Session::rowToSession($row);
    }
    return $sessions;
}
```

```

}
function getSessionsByCourse($course) {
    $query = "SELECT * FROM SESSION WHERE COURSE_ID = :course_id ORDER BY START";
    $stmt = Connection::getInstance()->prepare($query);
    $stmt->bindParam(':course_id', $course->id);
    $stmt->execute();
    $sessions = array();
    while ($row = $stmt->fetchObject()) {
        $sessions[] = Session::rowToSession($row);
    }
    return $sessions;
}

function getSessionsByTeacher($teacher) {
    $query = "SELECT * FROM SESSION WHERE TEACHER_ID = :teacher_id ORDER BY START";
    $stmt = Connection::getInstance()->prepare($query);
    $stmt->bindParam(':teacher_id', $teacher->id);
    $stmt->execute();
    $sessions = array();
    while ($row = $stmt->fetchObject()) {
        $sessions[] = Session::rowToSession($row);
    }
    return $sessions;
}

function update($session) {
    $query = "UPDATE SESSION SET TEACHER_ID = :teacher_id, COURSE_ID = :course_id,
        START = :start, END = :end, ADDRESS = :address, PRICE = :price WHERE ID = :id";
    $stmt = Connection::getInstance()->prepare($query);
    $stmt->bindParam(':teacher_id', $session->teacher_id);
    $stmt->bindParam(':course_id', $session->course_id);
    $stmt->bindParam(':start', $session->start);
    $stmt->bindParam(':end', $session->end);
    $stmt->bindParam(':address', $session->address);
    $stmt->bindParam(':price', $session->price);
    $stmt->bindParam(':id', $session->id);
    $stmt->execute();
    return true;
}

```

The SQL code uses prepared statements, a recent introduction in the PHP language. With the ease of use and data escaping, this kind of statements should be preferred for its readability and robustness, especially when maintenance is the priority. The update method shows an extensive use of such a statement.

We provide here 3 SQL methods for retrieving sessions. `getSessions` returns all the sessions found in the database. This method could be sufficient; if we want to filter on a teacher or a course, we can do it in a PHP loop. We can also let the filtering be done by the SQL engine, which performs best in tables with many data. We see how to do it in `getSessionsByTeacher` and `getSessionsByCourse`. To avoid code redundancy and ensure consistency between retrieval methods, we protect the row -to-object transfer in the `row ToSession` method.

The Connection object implements the singleton pattern, as a single connection is sufficient in our example, and reuses the Wordpress database connection:

```

class Connection {
    private static $instance;
    public static function getInstance() {
        if (!self::$instance) {
            self::$instance = new PDO(
                "mysql:host=" . constant("DB_HOST") . ";dbname=" . constant("DB_NAME"),
                constant("DB_USER"),
                constant("DB_PASSWORD"),
            );
        }
    }
}

```

```

        array(PDO::MYSQL_ATTR_INIT_COMMAND => 'SET NAMES ' . constant("DB_CHARSET"));
        self::$instance->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
    }
    return self::$instance;
}
}

```

The Course class is similar to the Person class as it reuses Wordpress data but the Wordpress API calls are different:

```

function getByMnemonic($mnemonic) {
    $course = new Course();
    $wp_post = get_posts(array('name' => $mnemonic))[0];
    $course->id = $wp_post->ID;
    $course->title = $wp_post->post_title;
    $course->description = $wp_post->post_content;
    return $course;
}

```

OO and Persistence Layers

As we have seen in writing the code for the various system classes, we introduce two new layers that encapsulate some of the complexity (Figure 6):

1. OO layer: classes, fields, and methods
2. Persistence layer: classes-to-DB mapping implementation (CMS and SQL code)

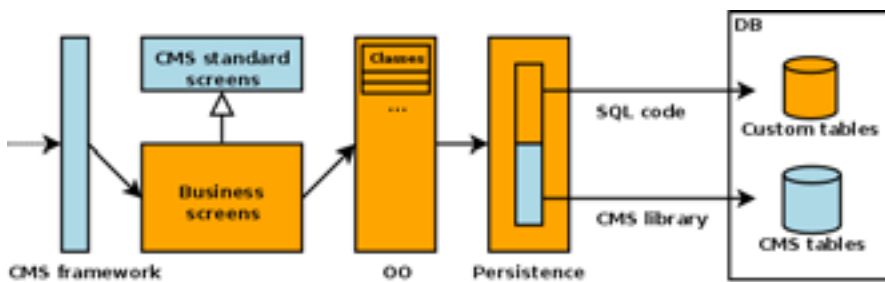


Figure 6. new architecture with OO and persistence layers

To implement these layers, we create two files OO.php and persistence.php. The OO file contains the system classes definitions: Person, Session, Course. The Persistence file contains the corresponding persistence implementations: PersonPersister, SessionPersister, and CoursePersister. Each persister class realizes the singleton pattern, as only one instance is needed:

```

class PersonPersister {
    private static $instance;
    public static function getInstance() {
        if (!self::$instance) {
            self::$instance = new PersonPersister();
        }
        return self::$instance;
    }
    // retrieve-and-storage functions
    // ...
}
// ... idem for SessionPersister and CoursePersister

```

As for an example of use, retrieving a teacher from its login can be coded like this:

```
$teacher = PersonPersister::getInstance()->getByLogin($login);
```

Code Readability

In Wordpress, we tweak the index.php theme page so the code can be accessed through the following address: *http://www.domain.com/?action=DisplaySessions*.

In the index page, the session display code is externalized into a separate PHP theme page:

```
if (isset($_REQUEST['action'])) {  
    include($_REQUEST['action'] . '.php');  
}  
else {  
    // ... standard index.php display code  
}
```

The externalization of the code into separate pages depending on the requested action realizes a simple MVC pattern (Figure 7):

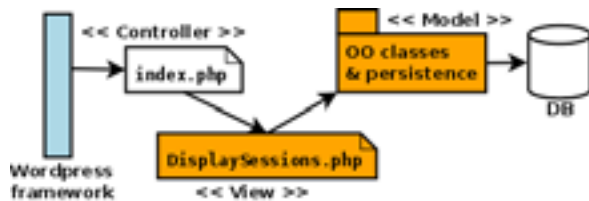


Figure 7. Simple MVC pattern in Wordpress

- The *controller* is the Wordpress framework passing the request to the index.php page, which in turn extracts the action request and its parameters to pass it to the corresponding page.
- The *view* is the page delivering the requested action; operations, data retrieval, and display.
- The *model* is the OO code layer, backed with its persistence layer and the database.

In the separate DisplaySessions.php page, the display code is minimal:

```
$sessions = SessionPersister::getSessions();  
foreach ($sessions as $session) {  
    // show session data in HTML  
}
```

Typical session data display presents the information in a rich HTML way, composed from the various session fields, in something like CSS-styled tables filled in with the useful information. Here is an example of table row:

```
<? $teacher = $session->getTeacher(); ?>  
<? $course = $session->getCourse(); ?>  
<tr class="session">  
    <td><a href="<?=$course->getDescriptionURL() ?>"><?=$course->title ?></a></td>  
    <td><a href="mailto:<?=$teacher->emailAddress ?>"><?=$teacher->firstName ?>  
    <?=$teacher->lastName ?></a></td>  
    <td><?=$formatDateTime($session->start) ?></td>  
    <td><?=$formatDateTime($session->end) ?></td>  
    <td><?=$session->address ?></td>
```

```
<td>${<?= $session->price ?></td>
</tr>
```

The table rows show the basic information for the students to take a course (Figure 8). The course title links to a detailed description of the course content. The link is provided in `getDescriptionURL()` by a Wordpress API call to `wp_permalink` – another Wordpress feature reuse. The teacher's name provides a link to send an e-mail, for example to get more information about the course and register for a session. The session start and end datetimes are formatted using the PHP facilities. The address where the session occurs may link to a Google Map for access directions. It is also possible to provide a Google Map below the table and show all the places where a session is planned. We do not interact with Google Map here but the Google Map API for PHP simply needs a few Session fields to display a map.

Flexible Feature Addition

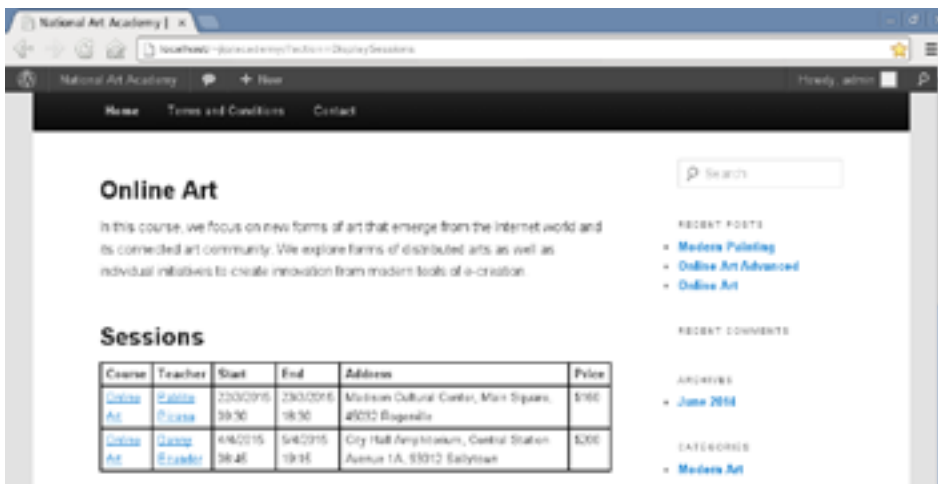
Adding features is also easy. For example, if we want to provide a session filter based on a particular course, the code needs little change:

```
$course = CoursePersister::getByMnemonic($_REQUEST['course']);
$sessions = SessionPersister::getSessionByCourse($course);
foreach ($sessions as $session) {
    // show session data in HTML
}
```

The HTTP request call requires the course reference as parameter:

`http://www.domain.com/?action=DisplaySessions&course=online-art`

The course mnemonic is passed as reference and is filtered through the Wordpress framework. The framework task structures the request and passes it to the `index.php` file, where our controller calls the `DisplaySessions.php` page. The first step in the page is to retrieve course data as an object from the persister. The session persister retrieves the sessions for the course, structured in an array that can be parsed to display the useful information in an HTML table (Figure 8):



Course	Teacher	Start	End	Address	Price
Online Art	E. Ponce	23/03/15 18:30	29/03/15 18:30	Modern Cultural Center, Main Square, 4000 Bogota	\$100
Online Art	E. Ponce	04/03/15 18:45	04/03/15 19:15	City Hall Art Museum, Central Station, Avenue 1A, 5012 Bogotá	\$200

Figure 8. Session display for a particular course

Unlayered Code

As a comparison with our approach, the unlayered code to access the Wordpress system and custom data could look like this:


```

if (isset($_REQUEST['action']) && ($_REQUEST['action'] == 'DisplaySessions')) {
    $connection = new PDO(
        "mysql:host=" . constant("DB_HOST") . ";dbname=" . constant("DB_NAME"),
        constant("DB_USER"), constant("DB_PASSWORD"),
        array(PDO::MYSQL_ATTR_INIT_COMMAND => 'SET NAMES ' . constant("DB_CHARSET")));
    $connection->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
    $query = "SELECT * FROM SESSION ORDER BY START";
    $stmt = $connection->prepare($query);
    $stmt->execute();
?>
<table><tr><th>Course</th><th>Teacher</th>
<th>Start</th><th>End</th><th>Address</th><th>Price</th></tr>
<?
    while ($row = $stmt->fetchObject()) {
?>
<tr class="session">
    <td><a href="<?= get_permalink($row->COURSE_ID) ?>">
        <?= get_post($row->COURSE_ID)->title ?></a></td>
    <td><a href="mailto:<?= get_userdata($row->TEACHER_ID)->user_email ?>">
        <?= get_user_meta($row->TEACHER_ID, 'first_name', true) ?>
        <?= get_user_meta($row->TEACHER_ID, 'last_name', true) ?></a></td>
    <? $start_dt = DateTime::createFromFormat("Y-m-d H:i:s", $row->START) ?>
    <? $end_dt = DateTime::createFromFormat("Y-m-d H:i:s", $row->END) ?>
    <td><?= $start_dt->format('d') . '/' . $start_dt->format('m') . '/' . $start_dt->format('Y')
        . ' ' . $start_dt->format('H') . ':' . $start_dt->format('i') ?></td>
    <td><?= $end_dt->format('d') . '/' . $end_dt->format('m') . '/' . $end_dt->format('Y')
        . ' ' . $end_dt->format('H') . ':' . $end_dt->format('i') ?></td>
    <td><?= $row->ADDRESS ?></td>
    <td><?= $row->PRICE ?></td>
</tr>
<?
}
?></table><?
}

```

The code is a mix of Wordpress framework structure, database connection, SQL code, Wordpress library calls, data formatting, and HTML display instructions. With the growing complexity of the application, code mix becomes less readable. In addition, changes in the Wordpress library, in the Wordpress database, or in the business database imply reviewing the entire code, which takes time and is prone to forgets and errors. While code mixing is fine for prototypes that remain so, separating the layers becomes necessary when writing a working web application, especially when it grows in complexity.

One Last Example: Adding A Google Map

Adding a feature like a Google map displaying the sessions for a course is quite easy and benefits from our approach as well. Using the same OO and persistence layers, the Google map code looks like this:

```
$course = CoursePersister::getByMnemonic($_REQUEST['course']);
$sessions = SessionPersister::getSessionByCourse($course);
foreach ($sessions as $session) {
    $teacher = $session->getTeacher();
    $map->addMarkerByAddress(
        $session->address, // locates the marker
        $teacher->firstName . ' ' . $teacher->lastName, // marker title
        '<a href="' . $course->getDescriptionURL() . '"' . $course->title . '</a><br>' .
            formatDateTime($session->start) . ' - ' . formatDateTime($session->end) . ' : ' .
            $teacher->firstName . ' ' . $teacher->lastName . '<br>' .
            $session->address); // detailed box
}
```

The code here shows what you need to feed to the Google Map API for PHP. Detailed code to add before and after the code above is routine and can be found in the sample code package. The specifics shown here are the `addMarkerByAddress` parameters required by the API. With a loop on the course sessions and calls to the OO objects, the data to display is easy to obtain. This shows how much our approach is simple yet powerful for more complicated features to add.

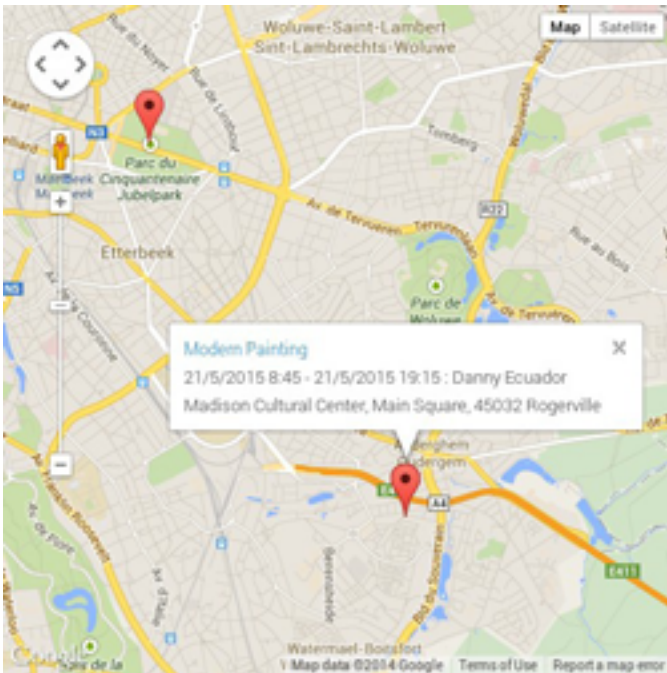


Figure 9. Google map displaying available sessions

The map shows up and centers on the course sessions to display. Here we show the example with two course sessions. In the map display code, the 3 `addMarkerByAddress` parameters behave as follows:

1. The marker locates the session address.
2. When the mouse is over the marker, a small title box – not shown on the picture – pops up and displays the teacher's name.
3. When the mouse clicks the marker, a detailed box appears and displays the course title with a link to the course description, the course start and end, the teacher's name and the session address.

Every feature we want to add are easy to write, read, and maintain. We could add a login box within the theme look and feel. Or we could add a session encoding form for the teachers. Our approach makes the code easy and natural.

Conclusion

As we have seen, CMS-based web applications allow rapid prototyping and reuse of CMS features like security management, database schema, and visual editors. However, integrating business code into the CMS can produce cumbersome, hard-to-maintain code. With the proposed approach that introduces OO and persistence layers, the business presentation code becomes clear, easy-to-maintain, and independent from the CMS. For example, Wordpress specifics are isolated into the persistent layer: any change made through Wordpress versions, or any kind of CMS replacement – even to another system like Drupal or SPIP, needs no business code change; everything to be changed can be found in a single layer. This isolation limits the number of changes to be made into the business presentation code and allows the presentation code to focus on business. Any kind of business change request or extension can be implemented into the focused code: given how simple, clear, and readable it is, change request implementation and bug-fix maintenance are made easy, fast, and reliable.

References

- Philippe Kruchten, The Rational Unified Process: An Introduction, Addison-Wesley professional, 2003.
- Grady Booch, James Rumbaugh, Ivar Jacobson, The Unified Modeling Language User Guide, Addison-Wesley professional, 2005. Kurt Bittner, Ian Spence, Use Case Modeling, Addison-Wesley professional, 2002.
- Ueli Wahli, Jean-Pierre Norguet, Jonas Andersen, Nicole Hargrove, Markus Meser, WebSphere Version 5 Application Development Handbook, IBM Redbooks, 2003: a lot about OO and Persistence (with J2EE)
- Wordpress Codex: <http://codex.wordpress.org>
- PHP Google Maps API: <http://bradwedell.com/php-google-maps-api>

About the Author



Jean-Pierre Norguet holds a Ph.D. in applied science and ICT from University of Brussels (ULB). In addition to several articles online, Dr. Norguet's publications include J2EE books with Prentice Hall and IBM Press, as well as several articles in international research conferences, the proceedings of which have been edited by major scientific organizations like ACM, IEEE, and Springer-Verlag. He contributed as a consultant, expert, and mentor to many ICT R&D projects, with a permanent strong concern about ethics. His other areas of interest include neuroscience-based stress management, peaceful interpersonal communication, and ecology-minded sobriety living.

Brand Integrity With Effective DevOps

by John Marx with Cigna and Capital One

Capital One, Cigna and Red Hat share high brand recognition and integrity. In the era of HIX Moments and Nasdaq Outages, we read in the daily news how ineffective IT processes can mar our companies' most treasured assets, our brands!

As business agility, cloud computing, social networks, and mobile computing, drive IT to an ever-increasing rate of software release, IT operations and development must become a unified force. Whether through dynamically provisioning stable development environments, or practicing continuous integration and delivery. Automation is a key to achieve better speed-to-market. This workshop and speakers' panel include leading professionals from three of our nation's premiere brands. Come hear and investigate how PaaS, IaaS, and SaaS are changing the way we work and how innovative technologies such as OpenShift and OpenStack are meeting this need. Effective DevOps includes flawless testing, security and governance. Red Hat platforms and Red Hat Consulting's agile processes meet the needs of today's fast paced IT while doing so in a flexible way.

Moderator

John Marx, CSM, CSP, SPC – Red Hat Consulting, Principal within the Agile Practice.

Speakers Panel

- Jason D. Valentino – Capital One Labs – Technology Innovation
- Curtis Yanko – Cigna Corporation, Architecture Manager – DevOps

This panel will provide concrete examples of how Red Hat technologies and agile delivery processes, increase speed-to-market, reduce project and business risk, sustain high quality, and embrace change.

Specific Topics include

- Cultural and Organizational prerequisites for effective DevOps
- Reducing Risk through Continuous Integration and Continuous Delivery
- Working toward a 'mature model' based on a Jez Humble's book titled Continuous Delivery
- Moving beyond perimeter hardening to prevent malware intrusion through third-party software libraries
- Prevention of Trojan malware through effective license management and use of resources such as Internet Crime Complaint Center (IC3) and NIST.org vulnerabilities lists
- Using the Scaled Agile Framework and establishing collaborative DevOps through a 'System Team'

About the Author



John Marx is a Services Delivery Manager and Principle Consultant within Red Hat's Agile Practice. He carries the Scrum Alliance's certifications such as Scrum Master, Scrum Professional and the Scaled Agile Framework's certification as a Scaled Agile Program Consultant.

Actualizing The Potential Shippable Increment

by John Marx

As Agile and DevOps gain popularity among enterprise organizations, the hype around these important disciplines increases. Discovering the true meaning of these terms and the practices they entail helps developers, architects and managers alike to thrive in their careers. Understanding how Agile development and Continuous Integration links with DevOps and Continuous Delivery is fundamental to reducing project risk and the on delivery expectations.

Closing the deal with the business requires the development teams not to just aim for the completion of code, but also to remain vested in the delivery processes to assure deployment. The steps to achieving business returns involve processes as simple as buying a used car. To avoid the jalopy, you must strike a bargain, test drive the vehicle, and close the deal.

At its onset in Snow Bird Utah agile was born by a developer-centric group of stakeholders. To their credit, they sought a better way and in their manifesto, they valued “Customer collaboration over contract negotiation”. But, the walls built between IT and the Business strike me as fortifications not built on tamped earth but instead, mistrust.

While agile pundits love to speak of ‘The Fall of Waterfall’ the silos within the enterprise remain safe storage for fodder. Fortunately, with DevOps, there is a hope that the integrated enterprise may yet flourish. When Continuous Integration stretches back to the customer with Acceptance Test-Driven Development and reaches forward to operations with Continuous Delivery, the potential of fast and frequent releases may be actualized into real *Return on Investment* (ROI).

I dream of CEO’s abdicating shareholder-centric planning which includes arbitrary annual planning cycles and myopic quarterly performance reviews for a system that is built on customer value and conscientious capitalism. While in its infancy, DevOps holds the promise of making this fantasy as a reality. To achieve this aim, three fundamental principles must be applied.

The Grand Bargain

The first principle which needs to actualize the potential of incremental delivery, is what I call “The Grand Bargain”. In an environment of mistrust formed through years of disappointments, a leap of faith is needed. The business had approached IT with amicable goals asking two questions in the past: How much? And How Long?

In reply to these, the pragmatic engineering types within IT reply with padded estimates and bloated budgets. The following round of the infamous used-car volley typically involves the haircut where the empowerment of IT begins with half the time and half the investment, yet no one has ever reduced the desired return nor the required scope.

Turning the delivery triangle upside down has been proposed as the means of achieving agility. This involves the assurance of predictable processes by fixing the time, schedule and floating scope. Agile relies heavily on the prioritization of backlogs to ensure that the need-to-haves are completed by the shortened project life cycle and that the nice-to-haves are the only stragglers that slip to release 2.0. This is a step in the right direction, but falls way short of a grand bargain.

The impediment is that most business stakeholders and customer proxies don’t remain engaged through iterations. So that, Sprint Planning truly prioritizes by business value. The reality that the Product Owner merely provides a contingent definition of done is a key hindrance to customer delight. All too often test results reviewed are seen only by analysts and developers, possibly some QA folk, but rarely the drivers of IT investment.

The Grand Bargain is the covenant that should be established at a project's onset when IT agrees to the haircut. A simple commitment by the project's sponsors to provide a viceroy for the ongoing planning and review will ensure that the scope is floating on a smooth sea. Naturally, this business representative must recognize that the demonstration of a working software is the pinnacle means of measuring progress, but also strip themselves of the plaid jacket and instead roll up the Oxford sleeves to invest in the alignment of incremental delivery with customer expectations. Only through this, the ongoing and iterative alignment and realignment may change to be embraced rather than resisting, and only by eliminating feature decay, we may secure our revenue realization and cost savings that remain paramount to Wall Street.

The Test Drive

No one would ever buy a used car without taking it for a spin. The risk of leaving the lot with a lemon is all too real. So why when making IT investment in software do so many business stakeholders end up with clunkers? I believe it is because people strive to develop software correctly rather than develop the correct software. Somewhere in our history of lobbing requirements documents over cubicle walls the fidelity gap with business requirements began and to bridge that gap someone needs to get out of their Aeron and spend some time watching demos.

But tests are scary. Sweaty palms and dry mouths abound whenever someone with the power to write checks enters the room. And just as a suitor on a blind date stumbles for words to articulate the ascetical beauty of their companion, software developers often wave their arms about gee-whiz features and better mousetraps leaving the pen-striped suitor from accounting wondering just what the heck they are looking at.

The Test Drive has to put the business stakeholder behind the wheel of the Sprint Demo. The driver is the business value and the alignment of features with the enablement of the customer. The how's focused upon becoming the all-too-important bridge between concepts and abstractions and true application and practice. It is no longer acceptable to speak of how the software works, but instead an in depth conversation about how the business works must ensue. As this dialogue continues the define-build-test teams become enlightened to the assumptions made by the customer and realize any gaps between how they think their software will be used and how the non-technical consumer of the invention will actually use it.

All of a sudden bits-and-bytes and speeds-and-feeds are replaced by the issues surrounding governance, risk management, audit and compliance. Organizational readiness and standard operating procedures take their proper place in the conversation and in depth and deliberate realities as to how and whether the software designed may actually be implemented becomes a key concern. To properly align delivery with expectation, a new stakeholder must join the team, namely operations! Hence DevOps the portmanteau of Development and Operations becomes the key practice to take Continuous Integration to Continuous Delivery. To do this, a new third principal must be practiced and that is credit approval.

Credit Approval and Closing The Deal

After the wife and kids ooh-and-ahh over the leather seats and great stereo Dad's palms start to sweat again because the next stop is the finance office. Soon, he is listing credit card accounts and social security numbers, knowing that a blue-suited credit manager is about to use cathode ray tubes to examine his life's work and sum it up in a single number, his FICO score. Only when this score is high enough, it will give the keys and perish the thought that errors in the credit report may need to be cleansed to drive the car home. Just like software, the detection of errors later in the process may delay closing the deal.

This step of determining whether matters are unrelated to the technical and functional performance of the car, a metaphor for our application, is analogous to the realities of operations and their impact on the drivability of our software, issues such as security for the first line of the contract. Risk management insists on knowing the business impact of the changes in procedure and the fault resilience that is needed to prevent the outage of critical business functions. Audit wishes to run parallel tests and balances to the penny the outcomes of complex logic and calculations. Compliance has a book of their own that only they know how to read and they must assure the team that no rules are broken and no regulations missed in our complex systems that remain yet to be used in production.

The whiz and whirl of these stakeholders' deliberations all too often are something engineers have no patience for. Too many times the brilliant developer comes off as mad scientist within this new cast of characters. This is why scaling agile in enterprises becomes a prerequisite when aligning the delivery of teams running in parallel. In many cases, this *Governance, Risk Management, Audit and Compliance* (GRAC) practice becomes a team of its own that must run in parallel and participate in the Scrum of Scrums or PSI Release Review and planning processes typically performed at the Program level.

Continuing to ignore these critical business reviews forces *User Acceptance Testing* (UAT) processes to find and remedy defects late in the development cycle causing unanticipated increases in cost and delays in schedule. Developers must not thump that chest at code completion and instead begin to invest in these downstream realities to truly deliver. Without these GRAC reviews the fixed points on our delivery triangle of time and cost now bend with the wind and our predictive process falls prey to reality after the developers thought it was time to go home.

Accommodating a realistic CI/CD (Continuous Integration/Continuous Delivery) lifecycle that includes GRAC and even automates many of these reviews prevents a glut of shelf-ware between development and operations. Even more, the revision of requirements and features brought about by this early understanding of implementation constraints allows a realistic prioritization of the true need-to-have's and eyes wide open by collaborative teams may enjoy mutual accountability that results in credit approval. The *Non Functional Requirements* (NFR) codified through effective architecture and the feature-driven process tailoring resulting from GRAC reduces time to market and risk while sustaining quality. Few *Software Development Life Cycle* (SDLC) frameworks offer so much for so little. Join the IT Revolution and transform unrealized potential through the disciplined practice of DevOps.

When credit is given we can all drive home in our shiny new system.

On the Web

- Red Hat and Test-Driven Development <http://www.redhat.com/consulting/enterprise-solutions/agile-development/test-driven-development.html>
- Red Hat Consulting's Agile Adoption Workshop <https://www.redhat.com/resources/library/datasheets/red-hat-agile-adoption-workshop-datasheet>

About the Author



John Marx is a Services Delivery Manager and Principle Consultant within Red Hat's Agile Practice. He carries the Scrum Alliance's certifications such as Scrum Master, Scrum Professional and the Scaled Agile Framework's certification as a Scaled Agile Program Consultant.

Languages in UIs

by Damian Czersnious

At some point, it might happen that your application will go worldwide and marketing people will tell you a story about reading in a native language. “Reading in Polish bounds Poles better to our service”. They’ll say that internationalization is a business requirement.

Application architecture should scream about business requirements. The application structure (folders and packages) should describe what the application can do. For example, `com.sanecoders.bakery.product.list` means that sanecoders.com owns bakery application that can present bakery products. A bad architecture describes what the application is made of, because it exposes the mechanisms instead of purpose. A *one.community.bakery.controllers* shows that *community.one* owns bakery application that has controllers. If there is a view package, there will be a big chance that the application is made of the *MVC* design pattern.

The *MVC* or any other design pattern is just a detail from the architectural point of view. It goes without saying that these patterns allow accomplishing certain goals, but they’re not a reason we create applications.

When we see a car, it’s pretty easy to guess its purpose. This is because we see its size, number of seats, the size of the empty space, etc. The first look at the car tells us nothing about patterns it’s made of. We should expect the same for the bakery application – to see the features (user stories) that create it.

There are also other details such as database (*OracleDb*, *CouchDb*), *UI* frameworks (*Swing*, *Vaadin*), application frameworks (*Java EE*, *Spring*), etc. They are all just tools and good architecture makes them easily exchangeable. The good architecture describes user stories.

A *UML* or any other helpful notation simplifies the design process, but it’s not an architecture. A drawing can’t serve services and after a while, it’s nowhere near how the code actually works. Dictionaries define architecture as a process of both planning and constructing, so it’s a final product that works in a thoughtful manner.

Architecture by Oxford Dictionary: “the conceptual structure and logical organization of a computer or computer-based system”. It doesn’t mean the logical organization of a drawing of the future system.

Keynote

Architecture decides on a place, on a name and a relationship between components. The code, package, folder or any file is a method of recording architecture. In a nutshell, architecture is a code. Architecture is a method of recording business requirements. A business requirement speaks through a very concrete architecture.

Based on that, the architecture of internationalization, which is a business requirement should describe it comprehensively. However, the mechanism that makes it operational, is a detail and shouldn’t be visible at first glance.

Architecture

Continuing with the bakery application example, we may introduce I18n interface. The I18n stands for internationalization, where the number 18 represents middle characters of that word. For more details, read the Glossary of W3C Jargon. To see whole code or run an application, please follow the steps:

- clone sources: `git clone https://bitbucket.org/sanecoders/bakery.git`
- navigate to bakery folder
- execute maven command: `mvn package jetty:run`
- navigate to: `http://localhost:8080`

```
package com.sanecoders.bakery.ui;
public interface I18n
{
    void setI18nGateway( I18nGateway i18nGateway );
    void updateTranslations();
}
```

The best place for the interface is `com.sanecoders.bakery.ui` package, which tells us that bakery application (owned by *sanecoders.com*) has some UI that, probably among other things, supports internationalization.

Every application page has to follow this interface. Although, its usage is optional. After all, it's just an interface. This supports extensibility without forcing and it's at the engineer discretion whether he or she wants to follow it. Let's move to the implementation.

```
package com.sanecoders.bakery.product.list.ui;
public class ProductListView implements I18n
{
    @Override
    void setI18nGateway( I18nGateway i18nGateway ) ( ... )
    @Override
    void updateTranslations() ( ... )
}
```

The `ProductListView` class on translation update call (`updateTranslation()` method) uses `I18nGateway` to translate its parts. The `I18nGateway` is a generic implementation of the translation mechanism – the detail.

This is how the `I18n` interface screams about available internationalization. The `ProductListView` class uses it, therefore it's expected to be a language sensitive. It seems to be obvious just after reading the class declaration. The `I18nGateway` class delivers translation mechanism, but without details on how it works. It just works.

```
package com.sanecoders.bakery.ui;
public class I18nGateway
{
    private final String i18nSourceName;
    public I18nGateway( String i18nSourceName )
    {
        this.i18nSourceName = i18nSourceName;
    }
    public String translate( String key, Locale locale )
    {
        return ResourceBundle.getBundle( i18nSourceName, locale ).getString( key );
    }
}
```

The mechanism is pretty simple. It uses *Java* `ResourceBundle` class to access available files that keep translations. This is a general design of the translation mechanism.

Comment

The bakery application is a small one and only one translation mechanism is needed. In case of more advanced solutions, `I18nGateway` should exist as an interface rather than an implementation class. It also should be easy to introduce such abstraction at any time, e.g. by extracting interface. This is what I call engineering awareness. On one hand, the solution is small enough without ambiguous structures and on the other hand, it's extensible. Keeping solution extensible is one of the fundamental unspoken client expectation that engineers need to meet when writing every line of the code. After all, people build things through extension.

Propagation of the architecture

The vast majority of the applications consist of many features that are grouped into the feature groups. The bakery application is not any different. The described ability to list bakery products is part of a product management component (`com.sanecoders.bakery.product`) that consists of list (`com...product.list`), add and edit (`com...product.edit`) features. What is more, it has also a user management component (`com.sanecoders.bakery.user`) that consists of a sign in (`com...user.login`), a sign out (`com...user.logout`) and language switch (`com...user.language`) features. Each of these functionalities (modules) plays a role of the independently deployable application unit. It's possible to plug in or plug out every module just by administrating the main pattern. It's a really good idea to maintain this status quo.

The possible solution is to provide internationalization within each module that will contain one translation file per each supported language. This is example of Polish language.

```
resources/com/sanecoders/bakery/product/list/ui/messages_pl_PL.properties
product.list.add = dodaj
product.list.edit = edytuj
product.list.save = zapisz
product.list.remove = usu\u0144
product.list.product.name = nazwa
product.list.product.price = cena
```

The platform, in our case bakery (`com.sanecoders.bakery.ui` package), advises modules about the supported languages. This is done with `SupportedL10ns` interface, where `l10n` stands for localization (read more about `w3c` jargon).

```
package com.sanecoders.bakery.ui;
public interface SupportedL10ns
{
    Locale POLISH = new Locale( "pl", "PL" );
    Locale ENGLISH = Locale.ENGLISH;
}
```

Now, it's time to advise UI components to use prepared translations.

```
package com.sanecoders.bakery.product.list.ui;
public class ProductListView implements I18n
{
    public static final String ADD_PRODUCT_ACTION_I18N_KEY = "product.list.add";
    public static final String EDIT_PRODUCT_ACTION_I18N_KEY = "product.list.edit";
    public static final String SAVE_PRODUCT_ACTION_I18N_KEY = "product.list.save";
    public static final String REMOVE_PRODUCT_ACTION_I18N_KEY = "product.list.remove";

    private I18nGateway i18nGateway;
    @Override
    public void setI18nGateway( I18nGateway i18nGateway )
    {
        this.i18nGateway = i18nGateway;
    }
    @Override
    public void updateTranslations()
    {
        if( Objects.nonNull( i18nGateway ) )
        {
            actions.forEach( ( action ) -> action.component.setCaption( i18nGateway.translate(
action.captionI18nKey, getLocale() ) ) );
        }
        productTable.updateTranslations();
    }
}
```

```
}
public void addToActionLayout( Component component, String captionI18nKey )
{
    if( actionLayout.getComponentIndex( component ) == NOT_EXISTING_COMPONENT )
    {
        actions.add( new Action( component, captionI18nKey ) );
        actionLayout.addComponent( component );
    }
}
(...)
```

The `ProductListView` class represents the bakery product list page. It provides the possibility to add actions through the `addToActionLayout (...)` method. The second argument is the caption internationalization key, which is used to find a concrete translation. It also provides predefined keys for actions such as add, edit, save and remove.

The last but not least, is to specify a place for the module translation resources.

```
package com.sanecoders.bakery.product.list.ui;
public final class ProductListI18nGateway extends I18nGateway
{
    public static ProductListI18nGateway create()
    {
        return new ProductListI18nGateway( "com.sanecoders.bakery.product.list.ui.messages" );
    }
    private ProductListI18nGateway( String i18nSourceName )
    {
        super( i18nSourceName );
    }
}
```

The `ProductListI18nGateway` class declares the translation resources in `.../product/list/ui` folder – especially using messages file.

Localization

With internationalization usually localization goes in a parallel. This is because localisation is an enabler for the translation mechanism. The `I18nGateway` API requires information about currently used localization to search for a proper translation.

Depending on a chosen UI framework, the localization mechanism may differ. The bakery application uses Vaadin, which carries such information within the UI object. Then, the information is propagated to the underlying objects e.g. buttons. To be honest, Vaadin 7 mechanism is a bit different. It's more static and requires additional implementation to be dynamic (see `L10nController` class and `enter()` method of the `ProductListView` class, which executes translation on the page enter), but essentially it works like this. At the end, `ProductListView` class, which extends Vaadin layout, may call simply `getLocale()` method to get desired information.

Reusability

Do we really have to follow internationalization architecture within each module? Maybe it's also fine to just handle it on an upper level where a group of modules belong to. For example, the product manager (`com.sanecoders.bakery.product`) is a platform for the bakery list and edit modules. Thus, maybe we can handle translation only there – one translation file for all modules of the product manager per each supported language.

We need to be careful with this assumption. Each module of the bakery application is a separately deployable unit. All associations, direct towards the platform. If one of the modules is to be part of the bigger family, it needs to follow family's rules. This is why the product manager decides e.g. on how the communications between underlying modules should look like. What is more, none of the modules should know each other and the platform should not know any details about modules too. For example, none of the classes from `com.sanecoders.bakery.product` knows anything about the content of the `com...product.list` or `com...product.edit` package. None of the classes from `com...product.list` knows anything about the content of the `com...product.list.ui` package.

Comment

Above doesn't seem to be true for `com...product.list.ui` and `com...product.list.ui.table` packages. Please, be careful when making some general patterns based on one example. There're many rules involved in decision about structuring packages. This is a different story.

This is a concept of the plug-inable architecture, where we can plug in or out modules and reduce or extend functionality. This also supports reusability, because we may reuse product list for different *UIs* or reuse the product manager for different product list or edit modules.

Handling internationalization on an upper level will break that design, because platform will define (know) details of the underlying modules. For example, above translation file for Polish language defines add, edit, save and remove translation keys for the product list module. However, on the product manager level, this file will define keys for the product edit module too. Consequences:

- Translation keys may contain errors, key names might be limiting, too few defined keys for the modules etc. This is a minor issue or a big one. All depends on how quick we can apply changes or who is the owner of the code: we or another company. Creating extensible solutions (as said before) is a fundamental unspoken user expectation. Unspoken is the synonym of non-functional requirement e.g. used in *Software Requirements Specification*.
- Modules do not necessarily have to know about these globals, which will result in hard discovery of resource duplication.
- Violates encapsulation, as part of the module details (translation keys) are outside the module and can be changed from outside.
- Violates *The Common Closure Principle*, because classes (here classes and resources) are not packaged together.
- Violates *The Common Reuse Principle* for the same reason.

The answer is: yes, we really want to internationalize each module separately.

A not Recommended Solution

Occasionally, internationalization can be seen as an unimportant detail instead of an important business requirement. Let's ponder on it for a moment.

The idea is to extend each *Vaadin's* object with a translation mechanism (e.g. `LangButton`, `LangTextField`, etc) and close them together in a special `com.sanecoders.bakery.ui.vaadin` package.

This solution seems to have several defects. From now on, we have to maintain all extended objects that are sensitive to framework upgrades. Testing isn't clean, because we are forced to spy testing objects. This is the price we need to pay for customizing a 3rd party's framework.

Adding new functionality by the extending existing objects was largely used in the 80's. Such technique is called *Programming by difference*. It's quite useful e.g. in unit testing, but it has troublesome traps which grow with the code. As a result complex code drown many projects. That was an important lesson we got. For more details read "Working efficiently with Legacy Code" chapter 8 page 94 by Michael Feathers.

Vaadin Context

When using *Vaadin*, we may use *AppFundation* plugin to provide internationalization. This is a static solution where translation takes place during an object creation. In practice, this happens on *URL* enter or page reload. Therefore, it's not possible to update translations in working application e.g. via link.

In modern web application, it's quite "strange" e.g. to be logged out after reloading the page. This is exactly what happens after re-initialization (previous session object has been dropped). To prevent that, we may decide to run application in a preserve mode using `@PreserveOnRefresh` annotation. Unfortunately, such setting limits translation capabilities, because the mentioned plug-in does the job during creation (see `createField()` method of the `I18nForm` class).

From the architectural point of view, such solution hides existence of the internationalization in the application. We talked about this in the previous chapter.

The *AppFundation* plugin mechanism equivalence is mentioned earlier `I10Controller` class plus the trigger for translation, which is the `enter()` method of the View interface (see e.g. `ProductListView` class for reference). These two little things deliver the dynamic mechanism.

Summary

When we talk about internationalization, few things have to be covered:

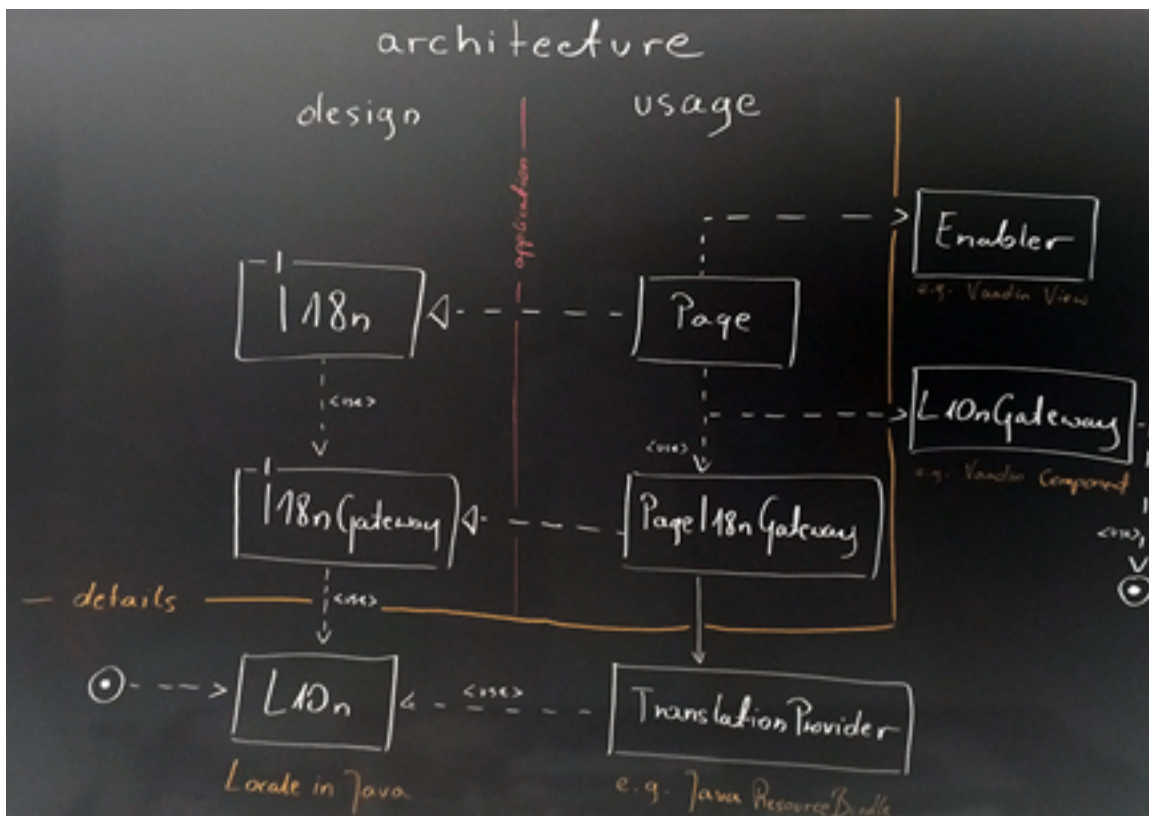


Figure 1. Internationalisation design

Architecture

- The way we say this page is language sensitive – `ProductListView` class implements `I18n` interface.
- The mechanism gateway for finding translations – `I18nGateway` class used with `I18n` interface.

Translation mechanism

- The mechanism of retrieving translations – Java `ResourceBundle` used within `I18nGateway` class.
- The storage for translations – messages files (consequence of using Java `ResourceBundle`).

Enabler point

- The localization information provider (determines the preferred language) – `I18nController` class updates localization.
- The translation runner – `ProductListView` implements Vaadin View's `enter()` method – invoked on page enter.

The translation mechanism and the enabler point are the details of the internationalization architecture. Both can be replaced with different frameworks. Java `ResourceBundle` e.g. with any database engine and *Vaadin* e.g. with another *UI* solution.

Solutions such as *AppFoundation* plug-in delivers translation mechanism, which bounds the application with technology. This is because they make shortcuts that can be seen an easy use (nice annotations, less code, greater transparency). Usage of this concrete plug-in removes internationalization architecture entirely from the application, leaving just mechanisms (translation and enabler). In case of moving away from *Vaadin*, the internationalization code will have to be changed instead of just replacing enabler point. To be honest such refactoring doesn't seem to be that difficult. It's just brainless and time consuming work. The problem is in thinking. If the internationalization was organized in that way, so other aspects are probably too. A brief look at such code can successfully kill motivation to act.

On Web

- Architecture by Oxford Dictionary, <http://www.oxforddictionaries.com/definition/english/architecture>
- Glossary of W3C Jargon, <http://www.w3.org/2001/12/Glossary#I18N>

About the Author

Damian Czerwinski is a software engineering coach at Nokia Networks, who wants to share his knowledge with others. He is passionate about reasoning in engineering. This is an example of his work. Feel free to comment on it!

Design Patterns in Perl – Part 1

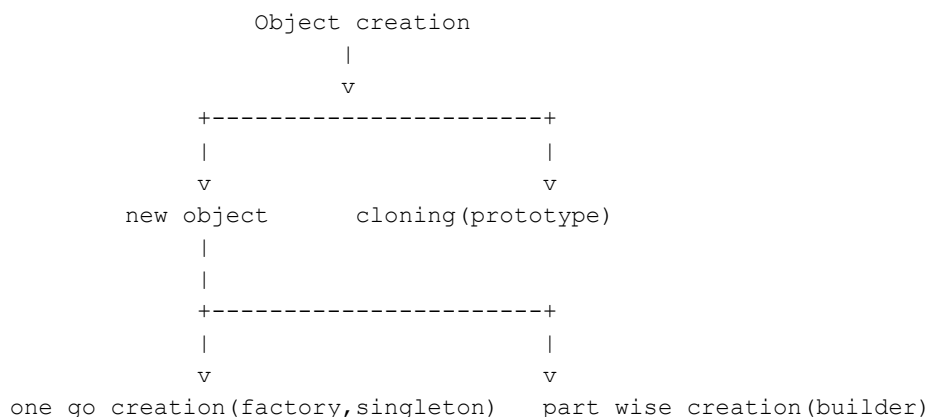
by Pravin Kumar Sinha

This is about designing in Perl. As Perl is an object oriented language, object oriented design in Perl is obvious. This document talks about the patterns used in Perl designing. A pattern is a basic unit of designing. When analysis phase is done the closest pattern is looked for the solution, or even some hybrid pattern also looked for a solution. Since patterns are well defined and tested way in solving the problem, once identified, can prove stable design and non modifiable code.

The more available patterns in different aspect of designing area supported in a language, more powerful that language is. In the document 23 GOF design patterns are discussed and implemented through Perl code. 23 design patterns are distributed among 3 categories. Creational, structural and behavioral. This document is written in the sequel and contains descriptions for creational patterns. The structure is in part 2 whereas behavioral in part 3.

Creational Design Patterns

Creational design pattern deals with creating the object, creating the component. A component is created or generated in two ways, generation from scratch (first of its kind) and generation from cloning. In the first way, again it can be generated in one go or part by part in a generic fashion. A factory class does new component creations where as builder does part by part (for intricate) while prototype does cloning.



Factory

Factory involves creating or generating new objects through methods. The method gives the concept of a factory which creates objects when provided, the id or name of the class. A soap factory creates soap. This is an object factory which creates objects. This method can be in product abstract class itself (factory method) or there can be an explicit factory class with method responsible for object creation (abstract factory).

Factory Method

Factory method creates an object through another method. These methods generally are available in libraries and calling methods in the library providing the objects. These methods can be thought as an interface in the library providing the objects. Factory methods can exist in the product abstract (interface) classes and passing class names, it can generate objects for implementation derived classes or Factory methods would be available with dedicated factory classes where one class is responsible for creating one particular product.

```

Static. . .      +-----+
factory      \    |   shape   |
method        .. .. .> |createshape(name|
                    |   :string):shape|
                    |draw():void   |
                    |enable():void  |
                    +-----+
                    / \
                    -
                    |<<extends>>
                    -----
                    |           |           |
+-----+ +-----+ +-----+
|   button   | |   menu   | |  combobox  |
+-----+ +-----+ +-----+
|draw():void | |draw():void | |draw():void |
|enable():void| |enable():void| |enable():void|
+-----+ +-----+ +-----+

```

Code Example

```

package shape;
use button;
use menu;
use combobox;
sub createshape {
my ($class,$string)=@_;
my $shape=undef;
if($string eq "menu") {
$shape = "menu";
}
if($string eq "button") {
$shape = "button";
}
if($string eq "combo") {
$shape = "combobox";
}
bless {}, $shape;
}
sub draw {
print "shape::draw\n";
}
sub enable {
print "shape::enable\n";
}
1;
<shape.pm>

package button;
use base qw(shape);
sub draw {
print "button:draw\n";
}
sub enable {
print "button::enable\n";
}
1;
<button.pm>

```

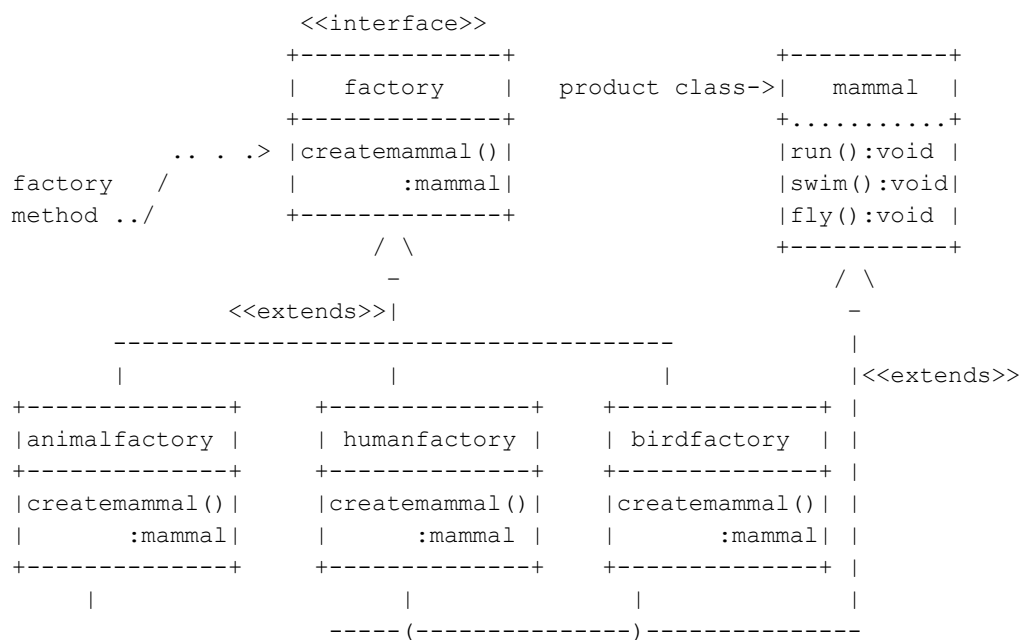
```
package menu;
use base qw(shape);
sub draw {
print "menu::draw\n";
}
sub enable {
print "menu::enable\n";
}
1;
<menu.pm>
```

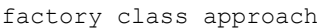
```
package combobox;
use base qw(shape);
sub draw {
print "combobox::draw\n";
}
sub enable {
print "combobox::enable\n";
}
1;
<combobox.pm>
```

```
use strict;
use warnings;
use shape;
shape->createshape("menu")->draw;
shape->createshape("button")->draw;
shape->createshape("combo")->draw;
<main.pl>
```

```
---data---
menu::draw
button:draw
combobox::draw
-----
```

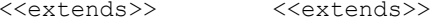
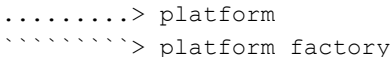
Class Approach





Abstract Factory

Abstract factory is similar to the factory class approach in factory method. Here, the product is supported on various platforms. It is a kind of two dimensional, set of products and each product on set of platforms. So as obvious, various platforms are the end objects to be created so factory would be one for each platform where factory methods will be for each product. When product and platform create orthogonal co-ordinates, factory classes and their Factory methods would be the same. Here the factory is known as Abstract as concrete, factories are platform specific.



```

| -----)-----
<<uses>> | \- - - | - - -<<uses>> - - \ |
| | v | | v |
| +-----+ +-----+ +-----+ +-----+
| |windowsbutton| |unixbutton| |windowsmenu| | unixmenu |
| +-----+ +-----+ +-----+ +-----+
| |draw():void | |draw():void| |draw():void| |draw():void|
| +-----+ +-----+ +-----+ +-----+
| ^ ^
| |
-----

```

Code Example

```

package abstractfactory;
sub new {
my $class=shift;
bless {},$class;
}
sub createmenu {
print "abstractFactory::createmenu\n";
}
sub createbutton {
print "abstractfactory::createbutton\n";
}
1;
<abstractfactory.pm>

```

```

package windowsfactory;
use base qw(abstractfactory);
use windowsmenu;
use windowsbutton;
sub createmenu {
windowsmenu->new;
}
sub createbutton {
windowsbutton->new;
}
1;
<windowsfactory.pm>

```

```

package unixfactory;
use base qw(abstractfactory);
use unixmenu;
use unixbutton;
sub createmenu {
unixmenu->new;
}
sub createbutton {
unixbutton->new;
}
1;
<unixfactory.pm>

```

```

package menu;
sub new {
my $class=shift;
bless {},$class;
}

```



```
1;
<menu.pm>
```

```
package button;
sub new {
my $class=shift;
bless {},$class;
}
sub draw {
print "button::draw\n";
}
1;
<button.pm>
```

```
package windowmenu;
use base qw(menu);
sub draw {
print "windowmenu::draw\n";
}
1;
<windowmenu.pm>
```

```
package unixmenu;
use base qw(menu);
sub draw {
print "unixmenu::draw\n";
}
1;
<unixmenu.pm>
```

```
package windowbutton;
use base qw(button);
sub draw {
print "windowbutton::draw\n";
}
1;
<windowbutton.pm>
```

```
package unixbutton;
use base qw(button);
sub draw {
print "unixbutton::draw\n";
}
1;
<unixbutton.pm>
```

```
use strict;
use warnings;
use windowfactory;
use unixfactory;
my $windowfactoryref= new windowfactory;
my $unixfactoryref= new unixfactory;
$windowfactoryref->createmenu->draw;
$windowfactoryref->createbutton->draw;
$unixfactoryref->createmenu->draw;
$unixfactoryref->createbutton->draw;
<main.pl>
```

```
---data---
```

```

windowmenu::draw
windowbutton::draw
unixmenu::draw
unixbutton::draw
-----

```

Singleton

As the name suggests, when only a single object of a class is created, the pattern we use to achieve this, is the singleton pattern. If the object creation function is used more than once, the already created object is returned, i.e. Someone needs to draw many coffee mugs and all to look equal but different in positions on the screen. So one object is enough as far as it looks concerned. Position can be adjusted externally. Here, factory would create a singleton object, in a province where the king can be one. All scalls to get the king would return the same king.

Code Example

```

use strict;
use warnings;
package singleton;
my $instance=undef;
sub new {
my $class=shift;
if(!defined $instance) {
$instance={};
bless $instance, $class;
} else {
return $instance;
}
}
package main;
print singleton->new, singleton->new, singleton->new;
<main.pl>

---data---
singleton=HASH(0x3e712c) singleton=HASH(0x3e712c) singleton=HASH(0x3e712c)
-----

```

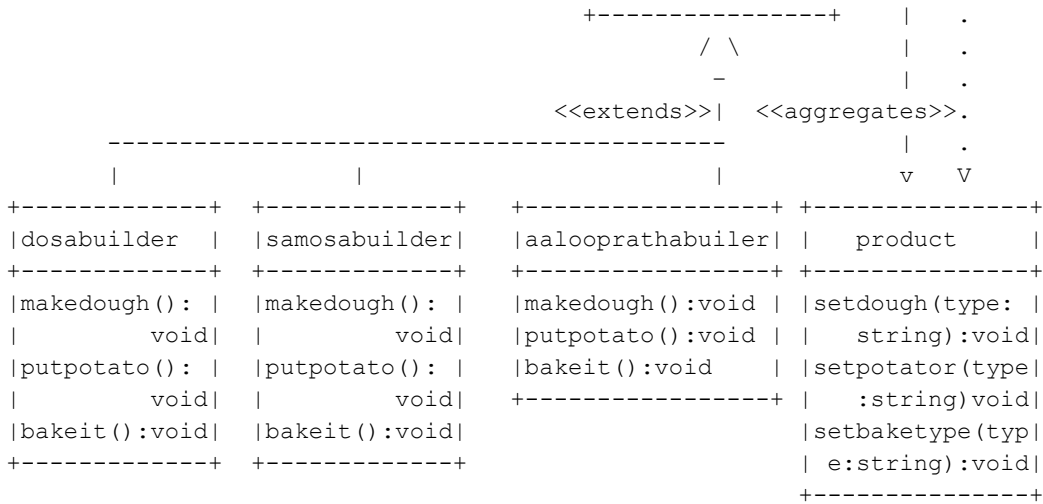
Builder Pattern

As the name suggests, the builder pattern is used when an entity (builder) creates a complex object part (predefined). Builder pattern exposes various methods in creating complex structures and it is the director who calls these methods and once the structure is ready, builder returns to the director. I.e in Indian restaurants, there can be food items to build, suach as Masala Dosa, Samosa, Aaloo Paratha. These items build through three different builders and these builders follow the same steps, a) Making dough b) Putting potato c) Baking it. Now, waiter who receives the order calls these three steps of builder without actually knowing the details. Once food item is ready, waiter returns the item to the customer.

```

. . . . .
.
+-----+ . +-----+ .
|waiter(director) | . . . . . |builder | .
+-----+ +-----+ <<uses>>
| | -----> |prodct:product | .
| | <<uses>> +-----+ .
|construct(builder| |makedough():void| .
| :builder):void| |putpotato():void|<---- .
|getproduct(): | |bakeit():void | | .
| product | |getproduct(): | | .
+-----+ | prodduct| | .

```



Code Example

```
package director;
sub new {
my $class=shift;
bless {},$class;
}
sub construct {
my ($ref, $builder)=@_;
$builder->makedough;
$builder->putaloo;
$builder->fry;
}
1;
<director.pm>

package masalabuilder;
use product;
sub new {
my $class=shift;
my $ref={};
${$ref}{product}=product->new;
bless $ref, $class;
}
sub getResult {
my $ref=shift;
$ref->{product};
}
1;
<masalabuilder.pm>

package dosabuilder;
use base qw(masalabuilder);
use product;
sub makedough {
my $ref=shift;
$ref->{product}->dough("wet");
}
sub putaloo {
my $ref=shift;
$ref->{product}->aloo("fried");
}
```

```
sub fry {
my $ref=shift;
$ref->{product}->fry("oil fry");
}
1;
<dosabuilder.pm>
```

```
package samosabuilder;
use base qw(masalabuilder);
use product;
sub makedough {
my $ref=shift;
$ref->{product}->dough("dry");
}
sub putaloo {
my $ref=shift;
$ref->{product}->aloo("fried");
}
sub fry {
my $ref=shift;
$ref->{product}->fry("deep fry");
}
1;
<samosabuilder.pm>
```

```
package product;
sub new {
bless {}, "product";
}
sub dough {
print "dough : ", $_[1], "\n";
}
sub aloo {
my ($ref, $string)=@_;
print "aloo : ", $string, "\n";
}
sub fry {
my ($ref, $string)=@_;
print "fry : ", $string, "\n";
}
1;
<product.pm>
```

```
use strict;
use warnings;
use director;
use samosabuilder;
use dosabuilder;
my $directorref=director->new;
$directorref->construct(new dosabuilder);
$directorref->construct(new samosabuilder);
<main.pl>
```

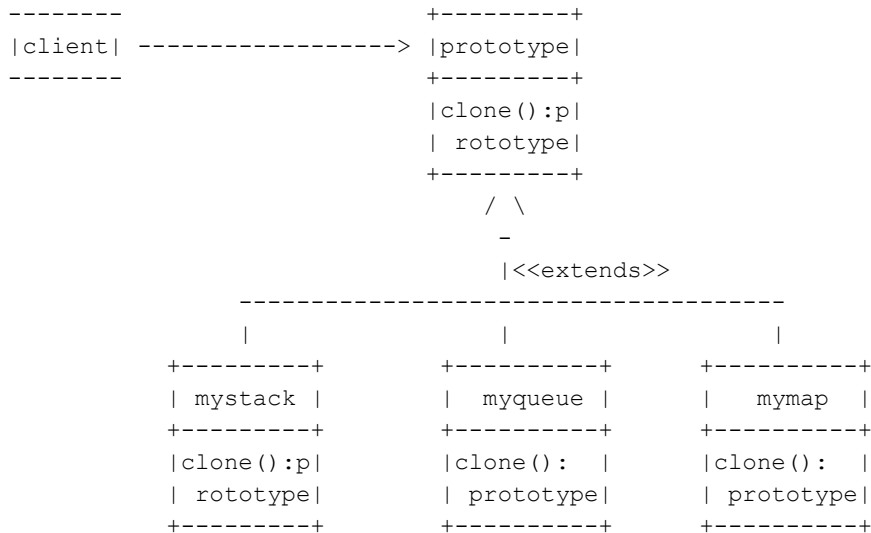
```
---data---
dough : wet
aloo : fried
fry : oil fry

dough : dry
```

```
aloo : fried
fry : deep fry
```

Prototype Pattern

When the cloning of an object is required, this pattern is used and when an object is created, it modifies its state and other data structures. When a new object carries the requirement of having a copy of the current state of an object, object has to clone itself. Every implementation class overrides abstract class clone function. A guru makes shishya (students), those are clones of him. They carry the same knowledge and guru does not lose anything.



Code Example

```
package dataobject;
sub new {
my ($class, $ref)=@_;
bless $ref,$class;
}
1;
<dataobject.pm>

package mystack;
use base qw(dataobject);
sub new {
my $ref=shift;
$ref->SUPER::new([@_]);
}
sub clone {
my $ref=shift;
(ref($ref))->SUPER::new([@$ref]);
}
1;
<mystack.pm>

package myqueue;
use base qw(dataobject);
sub new {
my $class=shift;
$class->SUPER::new([@_]);
}
sub clone {
my $ref=shift;
```

```
(ref($ref))->SUPER::new([@$ref]);
}
1;
<myqueue.pm>
```

```
package mymap;
use base qw(dataobject);
sub new {
my $class=shift;
$class->SUPER::new({@_});
}
sub clone {
my $ref=shift;
(ref($ref))->SUPER::new({%$ref});
}
sub printdata {
my $ref=shift;
my ($key, $value) = each %$ref;
print $key,"=>", $value;
while (((($key, $value)=each %$ref) && print "\",", $key,"=>", $value){});
}
1;
<mymap.pm>
```

```
use strict;
use warnings;
use dataobject;
use mystack;
use myqueue;
use mymap;
print "cloned stack data : ",@{mystack->new(1,2,3,4,5)->clone},"\\n";
print "cloned queue entries : ",@{myqueue->new('a','b','c','d','e')->clone},"\\n";
print "cloned map entries : ";
mymap->new("a","apple","b","berry","c","cherry")->clone->printdata;
<main.pl>
```

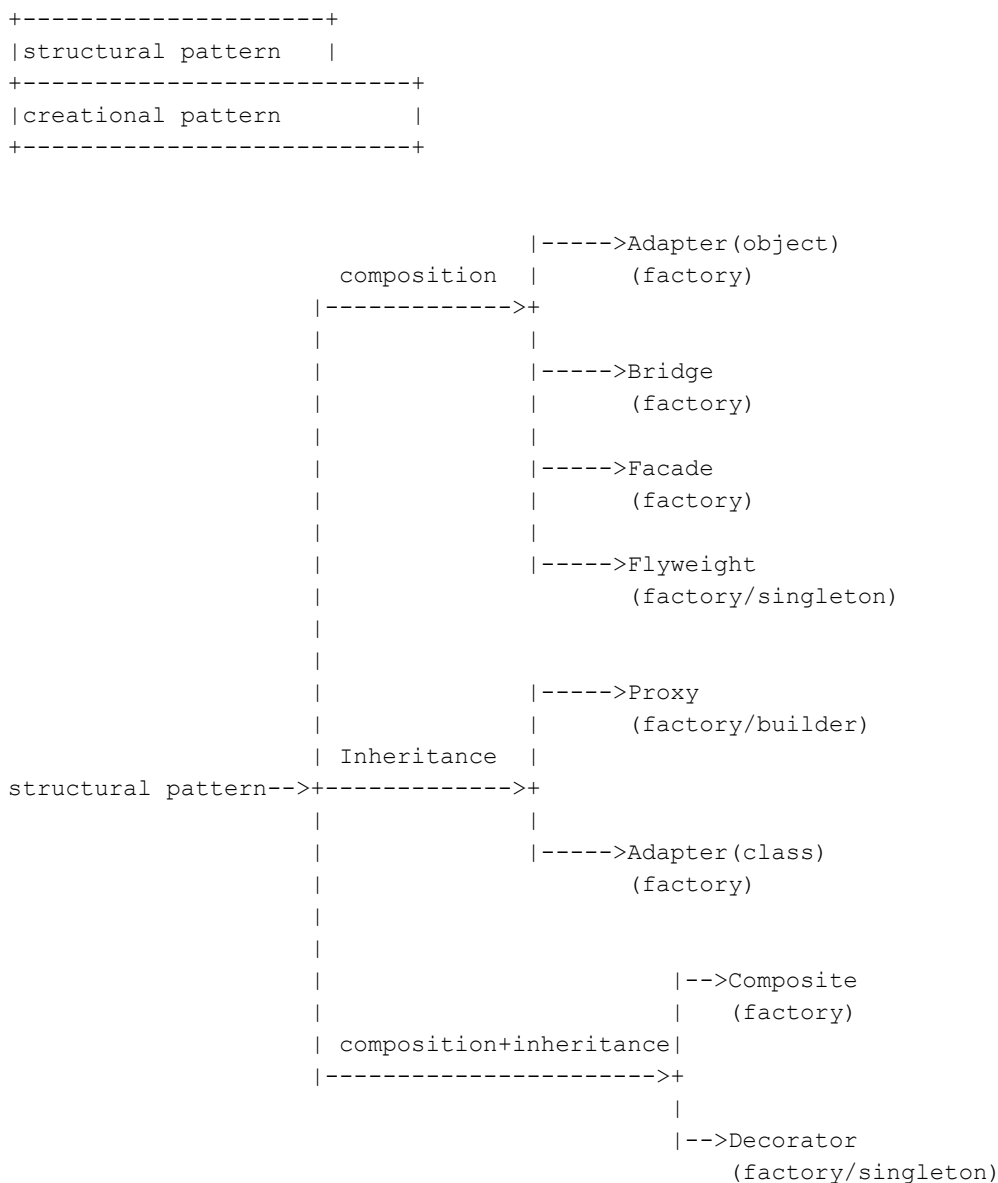
```
---data---
cloned stack data : 12345
cloned queue entries : abcde
cloned map entries : c=>cherry,a=>apple,b=>berry
-----
```

Design Patterns in Perl – Part 2

by Pravin Kumar Sinha

Structural patterns deals with the layout. Layout is the sense of how attributes of the class/ design are distributed and related. This deals with static structure of the design and not how the object interacts with each other. There can be many categories that can be distributed in. a) When a structure composes a different one. b) When a structure is derived from others so that it inherits and extends parent property. c) When a structure, composes other and inherits the same. Adapter (object), bridge, facade, flyweight are examples when a structure composes a different structure.

Proxy and adapter (class) are examples when one derives from another. Composite and decorator derive and compose the same. Apart from these every structural pattern has a base in one of the creational patterns. A structural pattern is based on a specific creational pattern. Similarly, behavioral pattern has a base on structural.



Composition

Structural patterns work in compositing other structures. First, structure which composes another structure, somewhere it delegates the client calls so in order to make use of it. Composed structure is generally passed to first one through the client. As this kind of pattern has primary structure and composed structure where primary structure delegates, the client calls to compose one, both client and primary structure keep the interfaces of next one and mix the code with interface methods. This makes the client and the primary structure code close for changes.

```

----- <<uses>> +-----+
|client| -----> |primarystrucInterface|<>--
-----
                    +-----+ |<<delegate calls to>>
                    / \      | +-----+
                    -      | --->|composedstrucinterface|
                    |      | +-----+
                    +-----+ / \
                    |Implementation| -
                    +-----+ |
                                +-----+
                                |implementation|
                                +-----+

```

Adapter

When adoptees interface varies from interface, a client expects this pattern to be used. It changes client supported interface to adaptee's interface and hence make the client use adaptee. An adapter can be of two types. First, 'object type' where adaptee interface is contained in adapter and any subclass of adaptee can be contained in the adapter at run Time whereas. The second one, is 'class type' where adaptee is also a base class for adapter. When a person talks to another one understanding different language, both have to use a multilingual two way adapter who let those two people communicate even though they do not understand each others language. Object type:

```

<<target interface>>
-----
|client| -----> | memory |
-----
                    +-----+
                    |fetchvideo(name|
                    | :string):void|
                    +-----+
                    / \
                    -
                    |
                    +-----+
<<adapter>>| memoryadapter |<>-----> | cameramemory |
                    +-----+
                    |adaptee:cameram|
                    | memory|
                    +-----+
                    |fetchvideo(name|
                    | :string):void|
                    +-----+
                    +-----+
                    / \
                    -
                    |
                    +-----+
                    | nickoncameramem |
                    +-----+
                    |setmode(mode:int):| . . .
                    | void|

```

```
|getfile(file:strlen|
|          g):void|
+-----+
```

Code Example

```
package memory;
sub new {
my ($class, $adaptee)=@_;
bless {adaptee=>$adaptee}, $class;
}
1;
<<memory.pm>>

package memoryadapter;
use base qw(memory);
sub fetchvideo {
my ($ref, $videoname)=@_;
$ref->{adaptee}->setmode(1);#read mode
$ref->{adaptee}->getfile($videoname);
}
1;
<<memoryadapter.pm>>

package cameramemory;
sub new {
my $class=shift;
bless {}, $class;
}
1;
<<cameramemory.pm>>

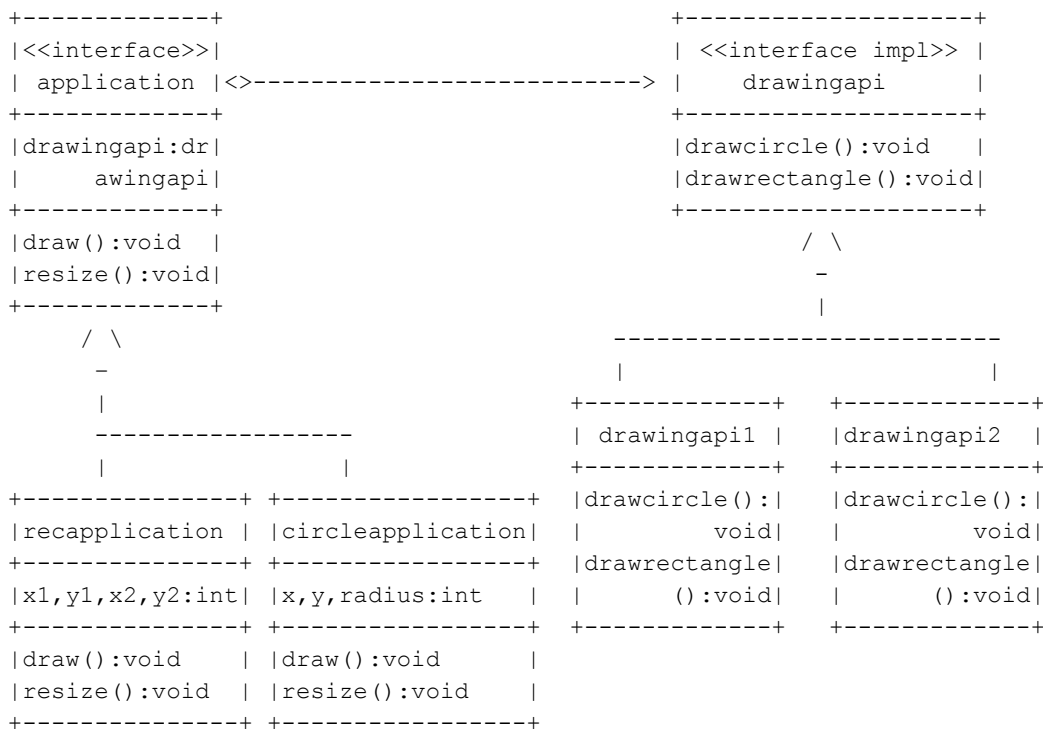
package nickoncameramem;
use base qw(cameramemory);
sub setmode {
my ($ref, $mode)=@_;
$ref->{MODE}=$mode;
if($ref->{MODE}) {
print "nickoncameramem::CHANGED TO READ MODE\n";
} else {
print "nickoncameramem::CHANGED TO WRITE MODE\n";
}
}
sub getfile {
my ($ref,$file)=@_;
return "ERROR in MODE\n" if !$ref->{MODE};
print "nickoncameramem::serving file : ",$file," :";
}
1;
<<nickoncameramem.pm>>

use memoryadapter;
use nickoncameramem;
memoryadapter->new(new nickoncameramem)->fetchvideo("earth song");
<<main.pl>>

---data---
nickoncameramem::CHANGED TO READ MODE
nickoncameramem::serving file : earth song :
-----
```

Bridge Pattern

When an abstract has several implementations and each implementation (say abstract implementation) grows again in a number of sub implementations then classes at sub implementation level exists in many numbers. Same set of sub implementations is available in each abstract implementation. It is important to separate abstract implementation tree with the sub implementation tree. This way abstract implementation and its sub implementation can grow independently. In a sub implementation tree we can consider that each sub implementation creates a bridge to an abstract implementation. Sub implementation is interfaced to the client, whereas abstract implementation is the real implementation which is delegated to four clients call to interfaces. For example, there are drawing APIs used across application types. We have all application types inherit all drawing APIs. If there are three application type supported and one new drawing API arrives, then three more inheritance is required. Better way to avoid that is to separate application hierarchy from drawing API hierarchy. Each application implementation will create a bridge to API implementation interface. In the country political system, there can be multiple parties and each party has a president, vice president, secretary, etc. Now number of party increases and also the posts. A person wants to be secretary in party A and another person wants to be president in the same party at. If we have an abstract class Post and then for each Post implementation all parties are subclasses then total number of parties classes would be too many. Here we can have Posts and Parties separate hierarchy and a person wants to become a Post holder in a Party will instantiate the party implementation passing the Post implementation. Like, new party A (new secretary). This way he becomes secretary in party A.



Code Example

```

package application;
sub new {
  my ($class,$ref)=@_;
  bless $ref,$class;
}
1;
<<application.pm>>

package recapplication;
use base qw(application);
sub new {
  my ($class, $x1,$y1,$x2,$y2, $drawingapi)=@_;

```

```
printf "recapplication::rectangle, x1=>%d, y1=>%d, x2=>%d, y2=>%d\n", $x1, $y1, $x2, $y2;
$class->SUPER::new({x1=>$x1, y1=>$y1, x2=>$x2, y2=>$y2, drawingapi=>$drawingapi});
}
sub resize {
my ($ref, $x1, $y1, $x2, $y2)=@_;
$ref->{drawingapi}->drawrectangle($ref->{x1}=$x1, $ref->{y1}=$y1, $ref->{x2}=$x2, $ref->{y2}=$y2);
}
sub draw {
my ($class, $x1, $y1, $x2, $y2)=@_;
$ref->{drawingapi}->drawrectangle($ref->{x1}, $ref->{y1}, $ref->{x2}, $ref->{y2});
}
1;
<<recapplication.pm>>
```

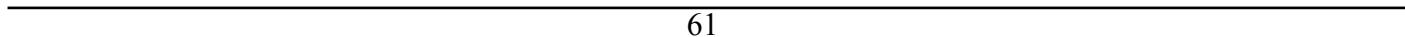
```
package circleapplication;
use base qw(application);
sub new {
my ($class, $x, $y, $radius, $drawingapi)=@_;
printf "circleapplication::circle, x=>%d, y=>%d, radius=>%d \n", $x, $y, $radius;
$class->SUPER::new({x=>$x, y=>$y, radius=>$radius, drawingapi=>$drawingapi});
}
sub resize {
my ($ref, $x, $y, $radius)=@_;
$ref->{drawingapi}->drawcircle($ref->{x}=$x, $ref->{y}=$y, $ref->{radius}=$radius);
}
sub draw {
my $ref=shift;
$ref->{drawingapi}->drawcircle($ref->{x}, $ref->{y}, $ref->{radius});
}
1;
<<circleapplication.pm>>
```

```
package drawingapi {
sub new {
bless {}, $class;
}
1;
<<drawingapi.pm>>
```

```
package drawingapi1;
use base qw(drawingapi);
sub drawrectangle {
my ($ref, $x1, $y1, $x2, $y2)=@_;
printf "drawingapi1::drawrectangle, x1=%d, y1=%d, x2=%d, y2=%d\n", $x1, $y1, $x2, $y2;
}
sub drawcircle {
my ($ref, $x, $y, $radius)=@_;
printf "drawingapi1::drawcircle, x=>%d, y=>%d, radius=>%d\n", $x, $y, $radius;
}
1;
<<drawingapi1.pm>>
```

```
package drawingapi2;
use base qw(drawingapi);
sub drawcircle {
my ($ref, $x, $y, $radius)=@_;
printf "drawingapi2::drawcircle, x=%d, y=%d, radius=%d", $x, $y, $radius;
}
sub drawrectangle {
```

Facade



Code Example

```
package calculator;
use subtracter;
use adder;
use multiplier;
use divider;
sub new {
my $class=shift;
bless {SUBTRACTER=>new subtracter,ADDER=>new adder,MULTIPLIER=>new multiplier,DIVIDER=>new
divider}, $class;
}
sub compute {
my ($ref, $expressionstring)=@_;
my ($operand1, $operand2);
if ($expressionstring=~/(.+)\+(.+)/) {
$operand1=$1;
$operand2=$2;
$ref->{ADDER}->compute($ref->compute($operand1),$ref->compute($operand2));
} elsif ($expressionstring=~/(.+)\-(.+)/) {
$operand1=$1;
$operand2=$2;
$ref->{SUBTRACTER}->compute($ref->compute($operand1),$ref->compute($operand2));
} elsif ($expressionstring=~/(.+)\*(.+)/) {
$operand1=$1;
$operand2=$2;
$ref->{MULTIPLIER}->compute($ref->compute($operand1),$ref->compute($operand2));
} elsif ($expressionstring=~/(.+)\/(.+)/) {
$operand1=$1;
$operand2=$2;
$ref->{DIVIDER}->compute($ref->compute($operand1),$ref->compute($operand2));
} else {
$expressionstring=~s/[ ]*([0-9]+)[ ]*/\1/;
$expressionstring;
}
}
1;
<<calculator.pm>>

package operator;
sub new {
my $class=shift;
bless {}, $class;
}
1;
<<operator.pm>>

package adder;
use base qw(operator);
sub compute {
my ($ref, $operand1, $operand2)=@_;
printf "adder::compute operand1=>%d, operand2=>%d \n",$operand1, $operand2;
eval($operand1 + $operand2);
}
1;
<<adder.pm>>

package subtracter;
```

```

use base qw(operator);
sub compute {
my ($ref, $operand1, $operand2)=@_;
printf "subtractor::compute operand1=>%d, operand2=>%d\n", $operand1, $operand2;
eval($operand1 - $operand2);
}
1;
<<subtractor.pm>>

package multiplier;
use base qw(operator);
sub compute {
my ($ref, $operand1, $operand2)=@_;
printf "multiplier::compute operand1=>%d, operand2=>%d\n", $operand1, $operand2;
eval($operand1 * $operand2);
}
1;
<<multiplier.pm>>

package divider;
use base qw(operator);
sub compute {
my ($ref, $operand1, $operand2)=@_;
print "divider::compute operand1=>", $operand1, " operand2=>", $operand2, "\n";
eval($operand1 / $operand2);
}
1;
<<divider.pm>>

use strict;
use warnings;
use calculator;
my $calculatorref=new calculator;
print "evaluation context : ", q(1-2+4*3-6/2+8*3-2*70/10), "\n";
my $result=$calculatorref->compute(q(1-2+4*3-6/2+8*3-2*70/10));
print "result : ", $result;
<<main.pl>>
---data---
evaluation context : 1-2+4*3-6/2+8*3-2*70/10
subtractor::compute operand1=>1, operand2=>2
multiplier::compute operand1=>4, operand2=>3
divider::compute operand1=>6 operand2=>2
subtractor::compute operand1=>12, operand2=>3
multiplier::compute operand1=>8, operand2=>3
divider::compute operand1=>70 operand2=>10
multiplier::compute operand1=>2, operand2=>7
subtractor::compute operand1=>24, operand2=>14
adder::compute operand1=>9, operand2=>10
adder::compute operand1=>-1, operand2=>19
result : 18
-----

```

Flyweight

When a class requires multiple instantiation and all have most of the properties same and few of them differs then its clever to instantiate only one object where as while rendering not in common properties among them would be provided externally. This kind of situation flyweight pattern is used. Objects common property is maintained only once in memory where as, properties different for each instantiation, passed from outside. The common intrinsic property saves a memory, whereas extrinsic property is passed only when required.

For example, in a housing colony map there are many houses in the map spread across various sectors and plots but houses would be in certain types only. Type A, type B, type C and type D (say). Each type has similar look and feel so the graphic data structure needs to be created only once for each type where same graphic would be shown in many places in the map and they only vary in space coordinates. Here houses types are intrinsic characteristics, whereas house location is extrinsic characteristics. An army officer does not know about each of soldier individually rather than he knows about different types of soldier groups (battalions) with different attires. Army officer just thinks about deploying a soldier groups to various places in war.



Code Example

```

package housetype;
sub new {
my ($class, $type)=@_;
bless {type=>$type},$class;
}
sub gethousetype {
my $ref=shift;
$ref->{type};
}
1;
<<housetype.pm>>

package house;
use base qw(housetype);
sub build {
my ($ref, $locationcontext)=@_;
print "house with type : ", $ref->gethousetype, " constructed at ", $locationcontext-
>get,"\n";
}
1;
<<house.pm>>

package locationcontext;
sub new {
my ($class, $sector, $plot)=@_;
bless {sector=>$sector, plot=>$plot}, $class;
}
sub get {

```

```

my $ref=shift;
qq(sector number : $ref->{sector}, plot number : $ref->{plot});
}
1;
<<locationcontext.pm>>

package housefactory;
use house;
sub new {
my $class=shift;
bless {},$class;
}
sub gethouse {
my ($ref, $type)=@_;
return $ref->{$type} if exists $ref->{$type};
$ref->{$type}=new house($type);
}
1;
<<housefactory.pm>>

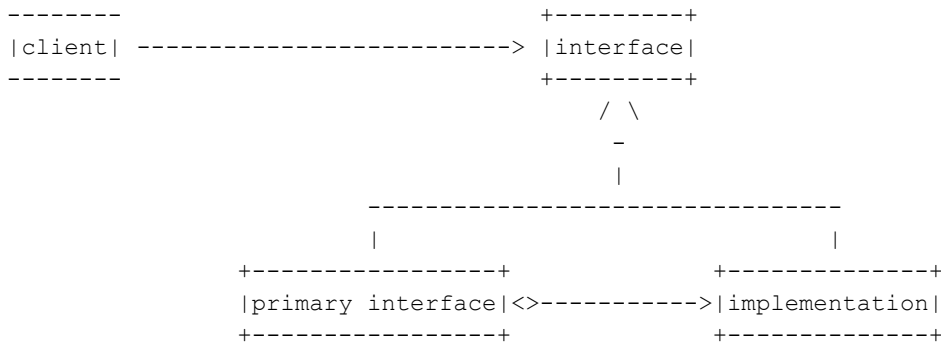
use strict;
use warnings;
use housefactory;
use locationcontext;

my $housefactoryref=new housefactory;
$housefactoryref->gethouse('A')->build(new locationcontext(10, 1));
$housefactoryref->gethouse('B')->build(new locationcontext(10, 2));
$housefactoryref->gethouse('A')->build(new locationcontext(10, 3));
$housefactoryref->gethouse('A')->build(new locationcontext(10, 4));
$housefactoryref->gethouse('B')->build(new locationcontext(10, 5));
$housefactoryref->gethouse('B')->build(new locationcontext(10, 7));
$housefactoryref->gethouse('C')->build(new locationcontext(11, 1));
$housefactoryref->gethouse('C')->build(new locationcontext(11, 2));
$housefactoryref->gethouse('C')->build(new locationcontext(11, 4));
$housefactoryref->gethouse('D')->build(new locationcontext(11, 5));
$housefactoryref->gethouse('D')->build(new locationcontext(11, 7));
<<main.pl>>
---data---
house with type : A constructed at sector number : 10, plot number : 1
house with type : B constructed at sector number : 10, plot number : 2
house with type : A constructed at sector number : 10, plot number : 3
house with type : A constructed at sector number : 10, plot number : 4
house with type : B constructed at sector number : 10, plot number : 5
house with type : B constructed at sector number : 10, plot number : 7
house with type : C constructed at sector number : 11, plot number : 1
house with type : C constructed at sector number : 11, plot number : 2
house with type : C constructed at sector number : 11, plot number : 4
house with type : D constructed at sector number : 11, plot number : 5
house with type : D constructed at sector number : 11, plot number : 7
-----

```

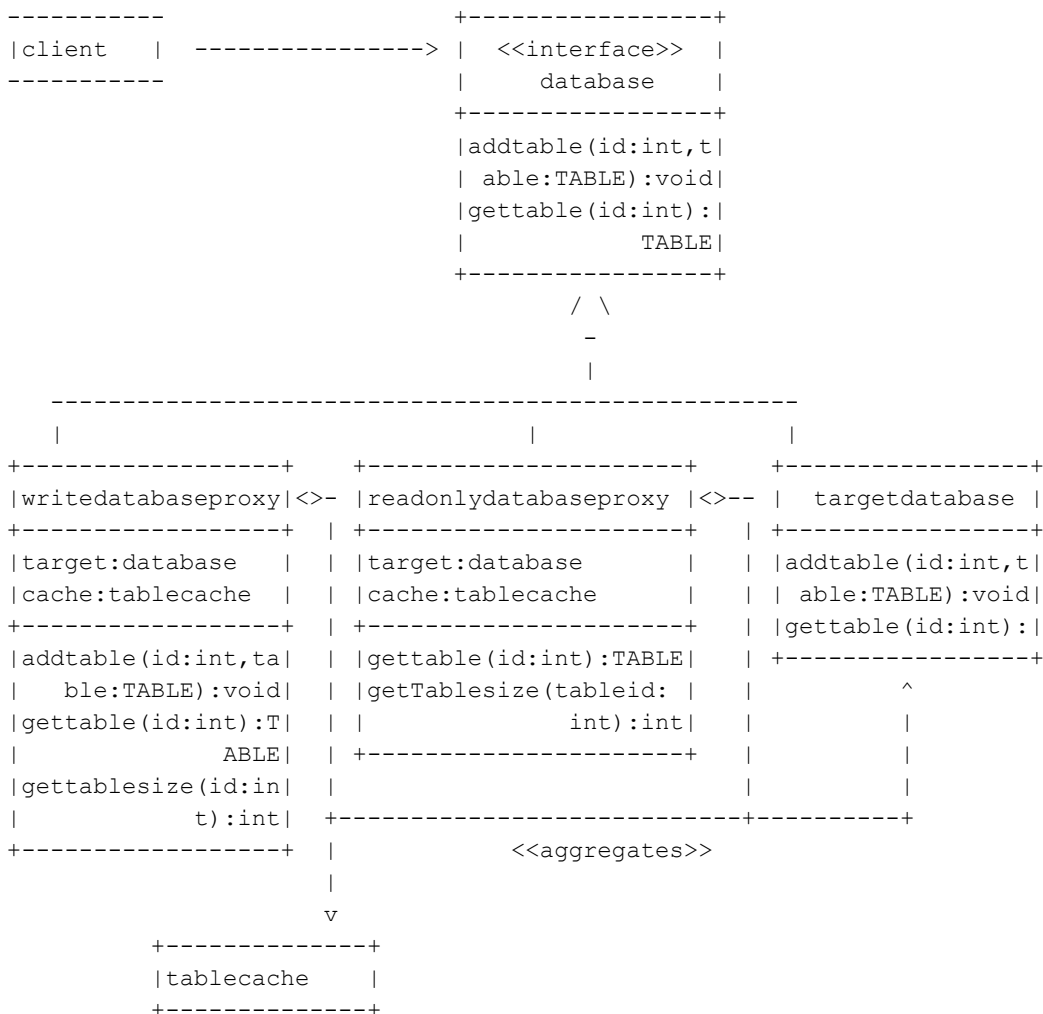
Inheritance

In this structural pattern classes grow their attributes through inheritance rather than composition. This makes attributes, static at run time. It is used when either primary interface and delegate both have same interface methods or primary interface implementation derives from delegate implementation, in this case delegate method would not extend.



Proxy

A class which acts as an interface to some other classes is a proxy. There can be a scenario when a target class is expensive to duplicate and a virtual class is required to be instantiated to many places, but in turn passes the call to target class of service. For example, a database program server requests based on a query. It brings database in memory in order to provide the service. Loading database is a costly operation and may not be possible at low end machines. So instantiating the database class would load the database. Since memory consumption is high, this class typically can be instantiated at high end servers only. Programs at other machines need a database service, can not instantiate database class. What is required here is a virtual database class that provides an exact same interface as the real database class, but internally keeps cached information and serves the client internally and when it is required, it contacts real database program instance for the service. For the client programs it is as if they instantiate real database programs (classes) only and use the same methods. Every country has an immigration department and some of them provide permission to stay in their country when you visit the country itself and they do not place their proxies in other countries.



```
|addtable(id:int|
|          t):TABLE|
|gettablesize(i|
|          d:int):int|
+-----+
```

Code Example

```
package database;
use targetdatabase;
use tablecache;
my $targetdatabaseref=undef;
my $tablecacheref=undef;
sub gettargetdatabase() {
  if (!defined $targetdatabaseref) {
    $targetdatabaseref=new targetdatabase;
  } else {
    $targetdatabaseref;
  }
}
sub gettablecache() {
  if (!defined $tablecacheref) {
    $tablecacheref=new tablecache;
  } else {
    $tablecacheref;
  }
}
sub new {
  my $class=shift;
  bless {},$class;
}
1;
<<database.pm>>
```

```
package tablecache;
sub new {
  my $class=shift;
  bless {},$class;
}
sub addtable {
  my ($ref, $tableid, $table)=@_;
  $ref->{$tableid}=$table;
}
sub gettable {
  my ($ref, $tableid)=@_;
  $ref->{$tableid};
}
1;
<<tablecache.pm>>
```

```
package writedatabaseproxy;
use base qw(database);
sub new {
  my $class=shift;
  my $ref=$class->SUPER::new;
  $ref->{TARGET}=$class->gettargetdatabase();
  $ref->{CACHE}=$class->gettablecache();
  $ref;
}
```

```

sub addtable {
my ($ref, $tableid, $table)=@_;
$ref->{TARGET}->addtable($tableid, $table);
$ref->{CACHE}->addtable($tableid, {SIZE=>scalar @$table,THRESHOLD=>0.8});
}

sub gettable {
my ($ref,$tableid)=@_;
$ref->{TARGET}->gettable($tableid);
}

sub gettablesize {
my ($ref, $tableid)=@_;
$ref->{CACHE}->{$tableid}->{SIZE};
}

1;
<<writedatabaseproxy.pm>>

package readonlydatabaseproxy;
use base qw(database);
sub new {
my $class=shift;
my $ref=$class->SUPER::new;
$ref->{TARGET}=$class->gettargetdatabase();
$ref->{CACHE}=$class->gettablecache();
$ref;
}

sub gettable {
my ($ref, $tableid)=@_;
$ref->{TARGET}->gettable($tableid);
}

sub gettablesize {
my ($ref, $tableid)=@_;
$ref->{CACHE}->gettable($tableid)->{SIZE};
}

1;
<<readonlydatabaseproxy.pm>>

package targetdatabase;
use base qw(database);
sub addtable {
my ($ref,$tableid,$table)=@_;
$ref->{$tableid}=$table;
}

sub gettable {
my ($ref, $tableid)=@_;
$ref->{$tableid};
}

1;
<<targetdatabase.pm>>

use strict;
use warnings;
use writedatabaseproxy;
use readonlydatabaseproxy;
my $wrdbtsproxyref=new writedatabaseproxy;
my $rddbtsproxyref=new readonlydatabaseproxy;
$wrdbtsproxyref->addtable("one",[[1,2,3],[4,5,6],[7,8,9]]);
$wrdbtsproxyref->addtable("two",[['a','b','c'],[['d','e','f'],[['g','h','i'],[['j','k','l']]]]]);
print "table size for tableid \"one\" : ",$rddbtsproxyref->gettablesize("one"),"\\n";
print "table data for tableid \"one\" :\\n";

```

```

map {print @{$_}, "\n"} @{$rddtbsproxyref->gettable("one")};
<<main.pl>>
---data---
table size for tableid "one" : 3
table data for tableid "one" :
123
456
789
table size for tableid "two" : 4
table data for tableid "two" :
abc
def
ghi
jkl
-----

```

Adapter class

Adapter class type serves the same idea as adapter object type. Adapter (and its subclasses) in addition to target are also derived from the adaptee. This makes the delegation easy where as sub classing adaptee is not possible.

```

<<target interface>>
-----+-----+
|client|----->| memory |
-----+-----+
|getvideo(name:| +-----+
| string):void| | cameramemory |
+-----+ +-----+
/ \ |getfile(name:st|
- | ring):void|
| |setmode(mode:in|
| | t):void|
| +-----+
<<extends>> / \
| -
| |
+-----+
|
+-----+
| memoryadapter |
+-----+
|getvideo():void|
|getfile():void |
|setmode():void |
+-----+
/ \
-
|
+-----+
| cameramemoryadapter|
+-----+
|getvideo():void |
|getfile():void |
|setmode():void |
+-----+

```

Code Example

```
package memory;
sub new {
my $class=shift;
bless {},$class;
}
sub getvideo {
my ($ref, $videoname)=@_;
print "memory::getvideo video name : ",$videoname,"\n";
}
1;
<<memory.pm>>

package memoryadapter;
use base qw(memory cameramemory);
sub getvideo {
my ($ref, $videoname)=@_;
print "memoryadapter::getvideo,video name : ",$videoname,"\n";
}
1;
<<memoryadapter.pm>>

package cameramemoryadapter;
use base qw(memoryadapter);
sub new {
my $class=shift;
my $ref=$class->SUPER::new;
$ref->setmode(0);
$ref;
}
sub getvideo {
my ($ref, $videoname)=@_;
$ref->setmode(1);
$ref->getfile($videoname);
}
1;
<<cameramemoryadapter.pm>>

package cameramemory;
sub new {
my $ref=shift;
bless {},$class;
}
sub setmode {
my $ref=shift;
$ref->{mode}=shift;
print "cameramemory::setmode, new mode : READ\n" if $ref->{mode};
}
sub getfile {
my $ref=shift;
if($ref->{mode}) {
print "cameramemory::getfile, getting file : ",shift;
}
}
1;
<<cameramemory.pm>>
```



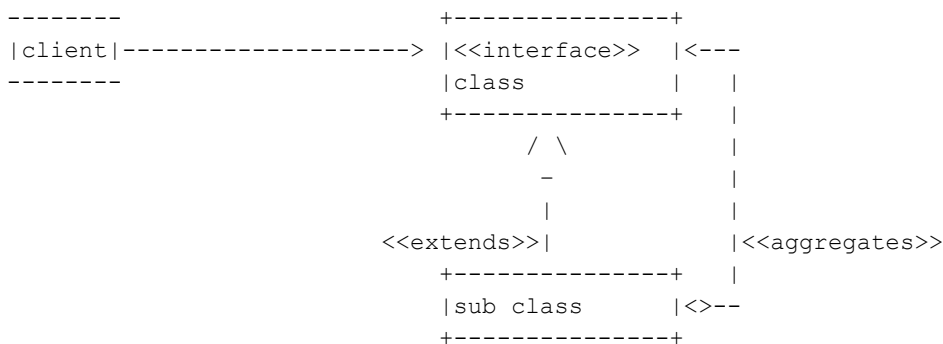
```

use strict;
use warnings;
use cameramemoryadapter;
cameramemoryadapter->new->getvideo("earth song");
<<main.pl>>
---data---
cameramemory::setmode, new mode : READ
cameramemory::getfile, getting file : earth song
-----

```

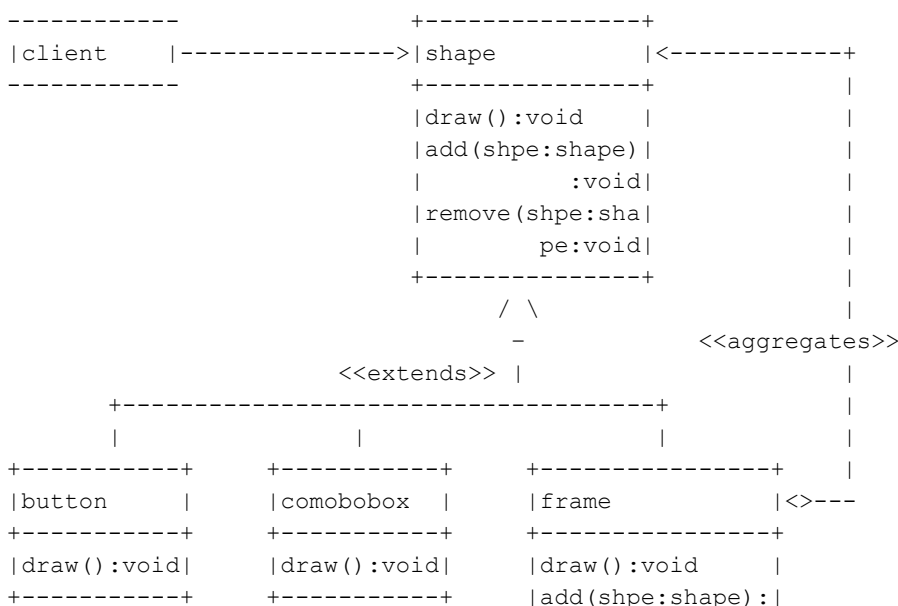
Composition+Inheritance

This kind of structural pattern, the structure consists of both inheritance and composition. sub class extending the base class also composes the base class. This way subclass provides same interface methods as base class and in addition it composes other leaf subclasses extending its attributes.



Composite Pattern

When a complex structure contains other structures, where other structures also provide the same interface behavior as complex structure. This situation makes it to composite structural pattern where structure extends its attribute through containing similar interface (subclassed from same parent) leaf classes. This becomes recursive in nature when complex structural subclasses. For example, a graphic component (i.e. Frame) can compose various other graphics (button, checkbox, frame) and out of those composed some composite graphic can contain similar objects (button, checkbox, frame). Here, the frame is a composite graphic which contains again leaf (button, checkbox) graphic and composite graphic (frame). In an arithmetic expression, operators are leaf elements, whereas expression itself is composite element. A code example is provided following graphic example.



```
|           void|
|remove(shpe:shap|
|           e:void|
+-----+
```

Code Example

```
package shape;
sub new {
my $class=shift;
bless [],$class;
}
1;
<<shape.pm>>
```

```
package frame;
use base qw(shape);
sub new {
my ($class, $frameid)=@_;
my $ref=$class->SUPER::new;
push @$ref, $frameid;
$ref;
}
sub draw {
my $ref=shift;
print "---\n";
printf "frame::draw, frameid:%s\n",${$ref}[0];
foreach my $child(@{$ref}[1..${$ref}]) {
$child->draw();
}
print "---\n";
}
sub add {
my ($ref, @childref)=@_;
push @$ref, @childref;
}
sub remove {
my ($ref, $childref)=@_;
delete @$ref[(grep{${$ref}[$_] =~/$childref/} 0..${$ref})];
}
1;
<<frame.pm>>
```

```
package button;
use base qw(shape);
sub new {
my ($class, $buttonid, $label)=@_;
my $ref=$class->SUPER::new;
push @$ref, $buttonid, $label;
$ref;
}
sub draw {
my $ref=shift;
printf "button::draw, buttonid:%s, button label:%s\n",${$ref}[0], ${$ref}[1];
}
1;
<<button.pm>>
```

```
package combobox;
```

```

use base qw(shape);
sub new {
my ($class, $comboboxid, @entries)=@_;
my $ref=$class->SUPER::new;
push @$ref, $comboboxid, @entries;
$ref;
}
sub draw {
my $ref=shift;
printf "combobox::draw, comboboxid:%s, entries:%s\n",${$ref}[0],join(' ',@{$ref}
[1..${$ref}]);
}
1;
<<combobox.pm>>

use strict;
use warnings;
use frame;
use button;
use combobox;

my $frametopref=new frame("top");
my ($frameleftref, $framerightref)=(new frame("left"), new frame("right"));
$frameleftref->add(new combobox("comboleft", "one", "two", "three"), new button("buttonleft",
"OK"));
$framerightref->add(new combobox("comboright", "animal", "bird", "reptile"), new
button("buttonright", "OK"));
$frametopref->add($frameleftref, $framerightref);
$frametopref->draw();
<<main.pl>>
---data---
---
frame::draw, frameid:top
---
frame::draw, frameid:left
combobox::draw, comboboxid:comboleft, entries:one two three
button::draw, buttonid:buttonleft, button label:OK
---
---
frame::draw, frameid:right
combobox::draw, comboboxid:comboright, entries:animal bird reptile
button::draw, buttonid:buttonright, button label:OK
---
---
-----

```

Example 2 (arithmetic expression)

```

-----+-----+
|client|-----> |<<interface>> |<-----+
-----+
|expression      |
+-----+
|compute():void|
+-----+
          / \
          -
          <<extends>>|
-----+-----+
|         |         |         |         |

```

```

+-----+ +-----+ +-----+ +-----+ |      |
|adder   | |subtractor| |multiplier| |divider  | |      |
+-----+ +-----+ +-----+ +-----+ |      |
|compute()| |compute():| |compute():| |compute()| |      |
|   :void| |   void| |   void| |   :void| |      |
+-----+ +-----+ +-----+ +-----+ |      |
                                           |      |
                                           |      |
                                           +-----+ |
                                           |expression|<>--+
                                           +-----+
                                           |compute():|
                                           |   void|
                                           +-----+

```

Code Example

```

package expression;
sub new {
my ($class, @ref)=@_;
bless [@ref], $class;
}
sub compute {
my ($ref, $expression)=@_;
foreach my $child(@$ref) {
$expression=$child->compute($expression);
}
$expression;
}
1;
<<expression.pm>>

package adder;
use base qw(expression);
sub compute {
my ($ref, $expression)=@_;
while($expression=~/(.*?)([-]?[0-9]+)([+])([-]?[0-9]+)(.*)/) {
printf "adder:compute, operand1=>%d, operand2=>%d\n", $2, $3;
$expression=join('', $1, eval ($2 + $3), $4);
}
$expression;
}
1;
<<adder.pm>>

package subtracter;
use base qw(expression);
sub compute {
my ($ref, $expression)=@_;
while($expression=~/(.*?)([0-9]+)(-)([0-9]+)(.*)/) {
printf "subtracter::compute, operand1=>%d, operand2=>%d\n", $2, $3;
$expression=join('', $1, eval ($2 - $3), $4);
}
$expression;
}
1;
<<subtracter.pm>>

package multiplier;

```

```

use base qw(expression);
sub compute {
my ($ref, $expression)=@_;
while($expression=~/(.*) (\d+) [*] (\d+) (.*)/) {
printf "multiplier::compute, operand1=>%d, operand2=>%d\n", $2, $3;
$expression=join('',$1,eval ($2 * $3), $4);
}
$expression;
}
1;
<<multiplier.pm>>

package divider;
use base qw(expression);
sub compute {
my ($ref, $expression)=@_;
while($expression=~/(.*) (\d+) \[ / (\d+) (.*)/) {
printf "divider::compute, operand1=>%d, operand2=>%d\n", $2, $3;
$expression=join('',$1,eval ($2 / $3), $4);
}
$expression;
}
1;
<<divider.pm>>

use strict;
use warnings;
use expression;
use adder;
use subtracter;
use divider;
use multiplier;

my $expressionref=new expression(new divider, new expression(new multiplier, new
expression(new subtracter, new expression(new adder))));
printf "evaluation expression:%s\n",q(1+3/3*2-2+6/2/3-2);
my $result=$expressionref->compute('1+3/3*2-2+6/2/3-2');
printf "result:%d\n",$result;
printf "evaluation expression:%s\n",q(1-2+4*3-6/2+8*3-2*70/10);
$result=$expressionref->compute(q(1-2+4*3-6/2+8*3-2*70/10));
printf "result:%d",$result;
<<main.pl>>
---data---
evaluation expression:1+3/3*2-2+6/2/3-2
divider::compute, operand1=>3, operand2=>3
divider::compute, operand1=>6, operand2=>2
divider::compute, operand1=>3, operand2=>3
multiplier::compute, operand1=>1, operand2=>2
subtracter::compute, operand1=>2, operand2=>2
subtracter::compute, operand1=>1, operand2=>2
adder::compute, operand1=>1, operand2=>0
adder::compute, operand1=>1, operand2=>-1
result:0
evaluation expression:1-2+4*3-6/2+8*3-2*70/10
divider::compute, operand1=>6, operand2=>2
divider::compute, operand1=>70, operand2=>10
multiplier::compute, operand1=>4, operand2=>3
multiplier::compute, operand1=>8, operand2=>3
multiplier::compute, operand1=>2, operand2=>7

```

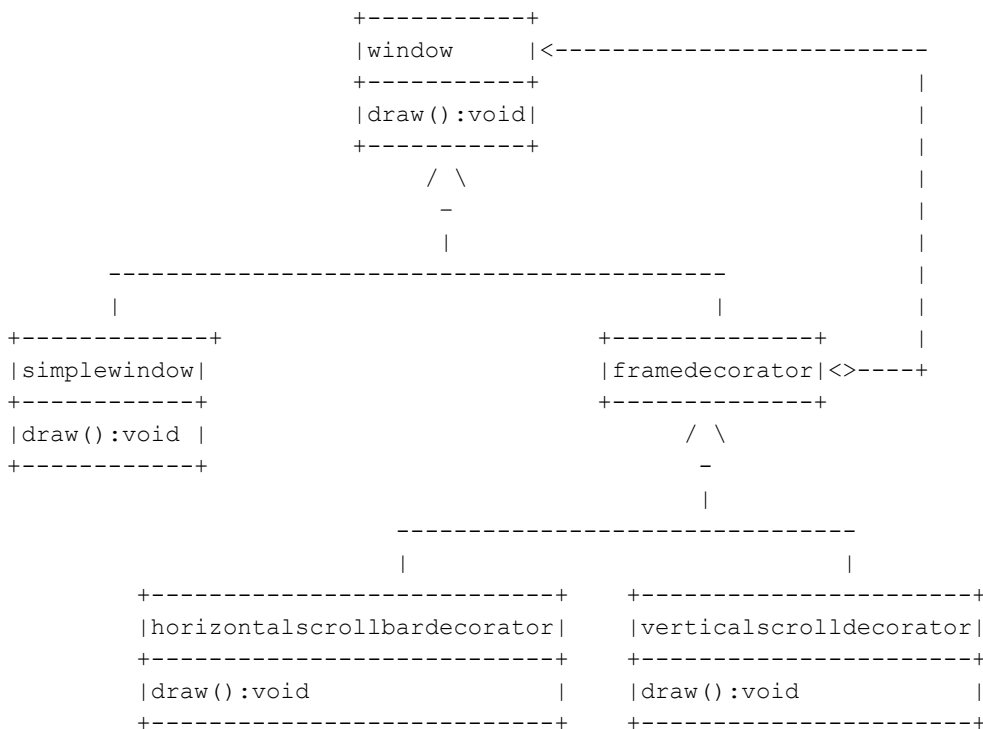
```

subtractor::compute, operand1=>1, operand2=>2
subtractor::compute, operand1=>12, operand2=>3
subtractor::compute, operand1=>24, operand2=>14
adder::compute, operand1=>-1, operand2=>9
adder::compute, operand1=>8, operand2=>10
result:18
-----

```

Decorator

When extending attributes is required at run time, a similar pattern to composite one works. Instead of the leaf class, it is the class for which attributes have to be extended and in place of composite, it is decorated classes which extend the attributes of the class. A decorator is further subclassed in order to have more attributes in. For example, of a leaf class window, a decorator class can be a frame window drawing frame around it, which is then subclasses to vertical scrollbar which provides a vertical scroll bar for the frame. A house knows how to show itself but, when it gets lawns and swimming pool around it as decorators, it looks elegant.



Code Example

```

package window;
sub new {
my $class=shift;
bless {}, $class;
}
1;
<<window.pm>>

package simplewindow;
use base qw(window);
sub draw {
print "simplewindow";
}
1;
<<simplewindow.pm>>

package framedecorator;

```

```

use base qw(window);
sub new {
my ($class, $decoratee)=@_;
my $ref=$class->SUPER::new;
$ref->{DECORATEE}=$decoratee;
$ref;
}
sub draw {
print "frame on :";
$ref->{DECORATEE}->draw;
}
1;
<<framedecorator.pm>>

package verticalscrollbardecorator;
use base qw(framedecorator);
sub draw {
print "verticalscrollbar on ";
$ref->SUPER::draw;
}
1;
<<verticalscrollbardecorator.pm>>

use strict;
use warnings;
use window;
use veriticalscrollbardecorator;

new verticalscrollbardecorator(new framedecorator(new window))->draw;
<<main.pl>>
---data---
verticalscrollbar on :frame on :simplewindow
-----

```

About the Author

xxxxx

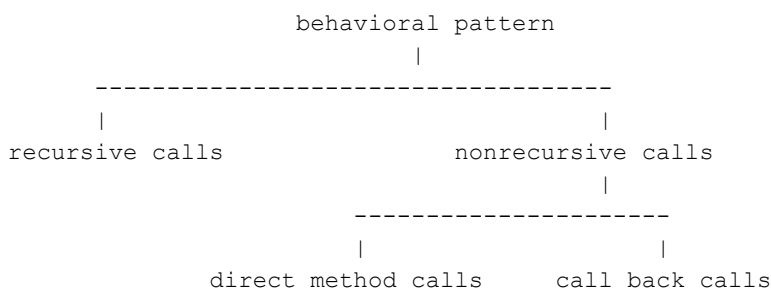
Behavioral design pattern includes patterns which focus on operations (activity) of a class. Every class/component is known for operations, it performs, the behavior of its attributes, rather than how it is structured or created.

```

+-----+
|behavioural pattern|
+-----+
|structural pattern  |
+-----+
|creational pattern  |
+-----+

```

For example, if we register a class method to a class and when a method of the class makes a callback to that function we call it command pattern. Another type can be when a class method (function) calls a recursively, same function of its own type of class object, it is a recursive way of calling the function and chain of responsibility is a pattern which matches to this. Behavioral patterns can be distributed in two main categories. a) Recursive method calls b) Non recursive method call. Non-recursive calls can be further distributed among Bi) direct method calls by) callback methods calls.



In the second example, arithmetic expression can also be solved through this pattern, code example I followed by this example. In another scenario a number can be shuffled. A code example is followed by the arithmetic expression example. In a football match goalkeeper passes the ball to the next player without actually knowing who will pass the ball in other side goalpost.


```

+-----+
|client| -----> +-----+
+-----+
|handlerrequest(| -----+
| request:reque|
| st):void      |
+-----+
/ \
-
|
+-----+
|          |          |          |
+-----+ +-----+ +-----+ +-----+
|stdoutcontroller| |stderrcontroller| |soundcontroller| |smscontroller|
+-----+ +-----+ +-----+ +-----+
| handlerrequest(r| | handlerrequest(r| | handlerrequest(| | handlerrequest(|
| equest:int):vo| | equest:int):vo| | request:int):| | t(request:i|
| id            | | id            | | void          | | in):void    |
+-----+ +-----+ +-----+ +-----+

```

Code Example

```

package handler;
sub new {
my ($class,$level)=@_;
bless {LEVEL=>$level}, $class;
}
sub next {
my $ref=shift;
if(scalar @_ ) {
$ref->{NEXT}=shift;
}
$ref->{NEXT};
}
sub handlerrequest {
my ($ref,$request,$level)=@_;
if($level>=$ref->{LEVEL}) {
$ref->handlerrequestimp($request);
$ref->next->handlerrequest($request,$level) if defined $ref->next;
}
}
1;
<<handler.pm>>

```

```

package stdoutcontroller;
use base qw(handler);
sub handlerrequestimp {
my ($ref,$request)=@_;
print "stdoutcontroller request:",$request,"\n";
}
1;
<<stdoutcontroller.pm>>

```

```

package stderrcontroller;
use base qw(handler);
sub handlerrequestimp {
my ($ref,$request)=@_;
print "stderrcontroller request:",$request,"\n";
}

```

```

1;
<<stderrcontroller.pm>>

package soundcontroller;
use base qw(handler);
sub handlerequestimp {
my ($ref, $request)=@_;
printf "soundcontroller request:%s\n",$request;
}
1;
<<soundcontroller.pm>>

package smscontroller;
use base qw(handler);
sub handlerequestimp {
my ($ref, $request)=@_;
printf "smscontroller request:%s\n",$request;
}
1;
<<smscontroller.pm>>

use strict;
use warnings;
use stdoutcontroller;
use stderrcontroller;
use soundcontroller;
use smscontroller;
(my $handlerref=new stdoutcontroller(1))->next(new stderrcontroller(2))->next(new
soundcontroller(3))->next(new smscontroller(4))->next(undef);
$handlerref->handlerequest("warning alert",1);
$handlerref->handlerequest("error alert",2);
$handlerref->handlerequest("major fault",3);
$handlerref->handlerequest("critical fault",4);
<<main.pl>>

---data---
stdoutcontroller request:warning alert
stdoutcontroller request:error alert
stderrcontroller request:error alert
stdoutcontroller request:major fault
stderrcontroller request:major fault
soundcontroller request:major fault
stdoutcontroller request:critical fault
stderrcontroller request:critical fault
soundcontroller request:critical fault
smscontroller request:critical fault
-----

```

Second example (arithmetic calculation)

```

-----+-----+
|client| -----> | handler | <---+
-----+-----+ |
|compute(expres| ----+
| sion):string |
+-----+
          / \
          -
          |

```

+-----+			
+-----+	+-----+	+-----+	+-----+
divider	multiplier	subtracter	adder
+-----+	+-----+	+-----+	+-----+
compute(expre	compute(expre	compute(expre	compute(expre
ssion:string	ssion:string	ssion:string	ssion:string
:void):void):void):void
+-----+	+-----+	+-----+	+-----+

Code Example

```

package handler;
sub new {
my ($class,$operator)=@_;
bless {OPERATOR=>$operator}, $class;
}
sub next {
my $ref=shift;
if (scalar @_ ) {
$ref->{NEXT}=shift;
}
$ref->{NEXT};
}
sub handle {
my ($ref,$expression)=@_;
print "expression:",$expression,"\n";
while($expression=~/(.*) ([-]?\d+) [$ref->{OPERATOR}] ([-]?\d+) (.*)/) {
$expression=join("",$1,eval qq($2 $ref->{OPERATOR} $3), $4);
}
if(defined $ref->next) {
$ref->next->handle($expression);
} else {
$expression;
}
}
1;
<<handler.pm>>

package divider;
use base qw(handler);
1;
<<divider.pm>>

package multiplier;
use base qw(handler);
1;
<<multiplier.pm>>

package subtracter;
use base qw(handler);
1;
<<subtracter.pm>>

package adder;
use base qw(handler);
1;
<<adder.pm>>

```

```

use strict;
use warnings;
use divider;
use multiplier;
use subtracter;
use adder;

(my $handlerref=new divider(q(/)))->next(new multiplier(q(*)))->next(new subtracter(q(-)))-
>next(new adder(q(+)))->next(undef);
print "result:",$handlerref->handle(q(1+3/3*2-2+6/2/3-2)),"\n";
print "result:",$handlerref->handle(q(1-2+4*3-6/2+8*3-2*70/10));
<<main.pl>>

---data---
expression:1+3/3*2-2+6/2/3-2
expression:1+1*2-2+1-2
expression:1+2-2+1-2
expression:1+0+-1
result:0
expression:1-2+4*3-6/2+8*3-2*70/10
expression:1-2+4*3-3+8*3-2*7
expression:1-2+12-3+24-14
expression:-1+9+10
result:18
-----

```

Reshuffling number:

```

-----+-----+
|client| -----> | handler | <---+
-----+-----+
|handle(numbe|
| rref:int*):| ----+
| void      |
+-----+
          / \
          -
          |
          +-----+
          |               |
+-----+ +-----+
| printer | | forwarder |
+-----+ +-----+
|handle(numbe| |handle(numbe|
| ef:int*):void| | ef:int*):void|
+-----+ +-----+

```

Code Example

```

package handler;
sub next {
my $ref=shift;
if(!defined $ref->{NEXT}) {
$ref->{NEXT}=(ref $ref)->new($ref->{POSITION}+1);
}
$ref->{NEXT};
}
sub new {

```

```

my $class=shift;
(scalar @_?bless {POSITION=>shift},$class:bless {POSITION=>1},$class);
}
1;
<<handler.pm>>

package digitelement;
use base qw(handler);
sub handle {
my ($ref,$tempref)=@_;
my $numberref=(ref $tempref?$tempref:\$tempref);
my ($startpos,$diffpos,$one,$two);
print "${$numberref}\n" if $ref->{POSITION} eq length($$numberref);
my $pos=$ref->{POSITION};
while($ref->{POSITION}<length($$numberref) && $pos<=length($$numberref)) {
$ref->next->handle($numberref);
do {
$startpos=$ref->{POSITION}-1;
$diffpos=++$pos-$ref->{POSITION};
$$numberref=~/.{ $startpos }(.{ $diffpos }) (.)?/;
($one,$two)=( $1,$2 );
}while defined --$diffpos && $one=~/.+ / && $two=~/.+ / && $one=~/$two/;
$$numberref=~s/(.{$startpos})(.){ $diffpos }(.)/\1\4\3\2/ if $pos<=length($$numberref);
}
$pos=$ref->{POSITION};
while($pos<length($$numberref)) {
$startpos=$pos+-1;
$$numberref=~s/(.{$startpos})(.){ $diffpos }(.)/\1\3\2/;
}
}
1;
<<digitelement.pm>>

use strict;
use warnings;
use digitelement;
new digitelement->handle(q(aad));
new digitelement->handle(q(1112411));
<<main.pl>>

```

```

---data---

```

```

aad
ada
daa
1112411
1112141
1112114
1114211
.
.
2111411
2114111
.
.
4111211
4112111
4121111
4211111
-----

```

Interpreter

In a language when representation of grammar is required along with an interpreter in order to decode sentences in language according to the grammar interpreter pattern is used. For example, in a language, there is the grammar / rules of Rahul AND (Abdul OR (Ravi OR John) AND sally. This grammar/rule says Rahul and sally and any of Abdul or rave or John. In case of a context, this grammar will be used by the interpreter in order to evaluate the context. When we hear a new word we interpret it through the dictionary.

```

-----
|context| -----> |interpreter| ----> |grammar|
-----

```

An interpreter makes reference to grammar in order to evaluate context.

Every religion has a grammar book.

```

-----
|client | -----> | context |
-----
|
|
+-----+
| expression | <-----+
+-----+
| interpret(c|
| ontext:str|
| ing):void |
+-----+
/ \
-
|
+-----+
|
|
+-----+ +-----+ +-----+
|terminalexpression| | OReexpression | | ANDexpression |
+-----+ +-----+ +-----+
| literal:string | | context:string | | context:string |
+-----+ +-----+ +-----+
| interpret():void | |interpret():void| |interpret():void|
+-----+ +-----+ +-----+
/ \ / \
\ / \ /
- -
| |
+-----+ +-----+

```

Code Example

```

package expression;
sub new {
my ($class,$ref)=@_;
bless $ref,$class;
}
1;
<<expression.pm>>

package terminalexpression;
use base qw(expression);
sub new {
my ($class,$literal)=@_;

```

```

my $ref=$class->SUPER::new({LITERAL=>$literal});
$ref;
}
sub interpret {
my ($ref,$context)=@_;
(scalar (grep {/$ref->{LITERAL}/} split(/(or)|(and)/i,$context)) ? 1:0);
}
1;
<<terminalexpression.pm>>

package orexpression;
use base qw(expression);
sub new {
my ($class,$expression1,$expression2)=@_;
my $ref=$class->SUPER::new({EXPRESSION1=>$expression1,EXPRESSION2=>$expression2});
$ref;
}
sub interpret {
my ($ref,$context)=@_;
$ref->{EXPRESSION1}->interpret($context) || $ref->{EXPRESSION2}->interpret($context);
}
1;
<<orexpression.pm>>

package andexpression;
use base qw(expression);
sub new {
my ($class,$expression1,$expression2)=@_;
my $ref=$class->SUPER::new({EXPRESSION1=>$expression1,EXPRESSION2=>$expression2});
$ref;
}
sub interpret {
my ($ref,$context)=@_;
$ref->{EXPRESSION1}->interpret($context) && $ref->{EXPRESSION2}->interpret($context);
}
1;
<<andexpression.pm>>

use strict;
use warnings;
use terminalexpression;
use orexpression;
use andexpression;

my $expressionref=new andexpression(new terminalexpression('RAHUL'),new orexpression(new
terminalexpression('ABDUL'),new andexpression(new orexpression(new terminalexpression('RAVI'),new
terminalexpression('JOHN')),new terminalexpression('UDONG'))));
print "grammar:RAHUL AND (ABDUL OR ((RAVI OR JOHN) AND UDONG))\n";
print "interpreting 'JOHN AND UDONG AND RAHUL':", $expressionref->interpret('JOHN AND UDONG
AND RAHUL'), "\n";
print "interpreting 'RAHUL AND RAVI':", $expressionref->interpret('RAHUL AND RAVI');
<<main.pl>>

---data---
grammar:RAHUL AND (ABDUL OR ((RAVI OR JOHN) AND UDONG))
interpreting 'JOHN AND UDONG AND RAHUL':1
interpreting 'RAHUL AND RAVI':0
-----

```

Arithmetic expression computation:

```

-----
|client|-----> |context|
-----
|
|
+-----> | expression | <-----+
+-----+
| interpret(con|
| text:string)|
| :string      |
+-----+
      / \
      -
      |
+-----+
|
+-----+ +-----+
|terminalexprssion| | andexpression |
+-----+ +-----+
| literal:string | | expression:string| <>-+
+-----+ +-----+
| interpret(context| | interpret(context|
| t:string):string| | :string):string |
+-----+ +-----+

```

Code Example

```

package expression;
sub new {
my ($class,$ref)=@_;
bless $ref,$class;
}
1;
<<expression.pm>>

package andexpression;
use base qw(expression);
sub new {
my ($class,$expression1,$expression2)=@_;
my $ref=$class->SUPER::new({EXPRESSION1=>$expression1,EXPRESSION2=>$expression2});
$ref;
}
sub interpret {
my ($ref,$context)=@_;
print "andexpression::context:",$context,"\n";
($context=$ref->{EXPRESSION1}->interpret($context)) && ($context=$ref->{EXPRESSION2}-
>interpret($context));
$context;
}
1;
<<andexpression.pm>>

package terminalexpression;
use base qw(expression);
sub new {
my ($class,$literal)=@_;
my $ref=$class->SUPER::new({LITERAL=>$literal});

```



```

$ref;
}
sub interpret {
my ($ref,$context)=@_;
print "terminalexpression::context:",$context,"\n";
while($context=~/(-?\d+)(($ref->{LITERAL}) (-?\d+)/) {
$context=join('',$`,eval "$1$2$3",$');
}
$context;
}
1;
<<terminalexpression.pm>>

use strict;
use warnings;
use terminalexpression;
use andexpression;
print "result:",new andexpression(new andexpression(new terminalexpression('/'),new
terminalexpression('\*')),new andexpression(new terminalexpression('-'),new
terminalexpression('\+'))->interpret('1+3/3*2-2+6/2/3-2'),'n";
print "result:",new andexpression(new andexpression(new terminalexpression('/'),new
terminalexpression('\*')),new andexpression(new terminalexpression('-'),new
terminalexpression('\+'))->interpret('1-2+4*3-6/2+8*3-2*70/10');
<<main.pl>>

---data---
andexpression::context:1+3/3*2-2+6/2/3-2
andexpression::context:1+3/3*2-2+6/2/3-2
terminalexpression::context:1+3/3*2-2+6/2/3-2
terminalexpression::context:1+1*2-2+1-2
andexpression::context:1+2-2+1-2
terminalexpression::context:1+2-2+1-2
terminalexpression::context:1+0+-1
result:0
andexpression::context:1-2+4*3-6/2+8*3-2*70/10
andexpression::context:1-2+4*3-6/2+8*3-2*70/10
terminalexpression::context:1-2+4*3-6/2+8*3-2*70/10
terminalexpression::context:1-2+4*3-3+8*3-2*7
andexpression::context:1-2+12-3+24-14
terminalexpression::context:1-2+12-3+24-14
terminalexpression::context:-1+9+10
result:18
-----

```

Non Recursive patterns

In this kind of pattern, a method calls other in linear(direct call) fashion or because of they are already registered (call back).

Callback Patterns

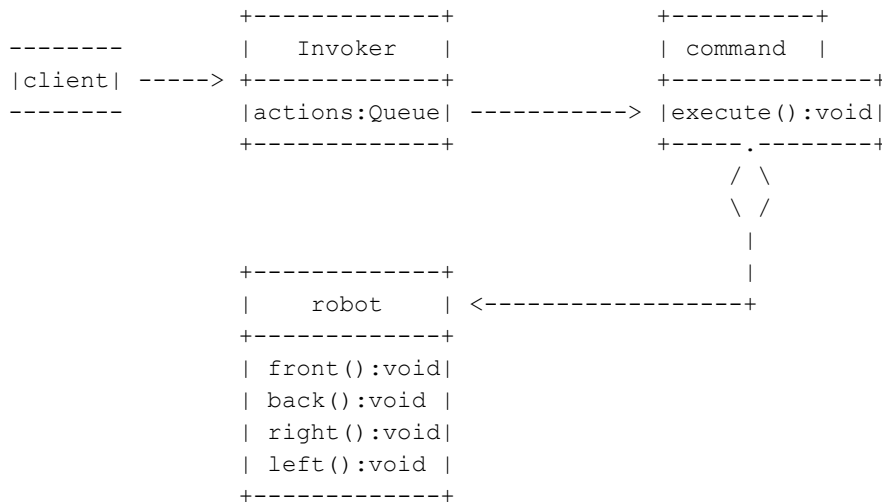
As per the name callee is already registered with caller and some event triggers callers to callback callee.

Command

When different methods that need to be called in, a generic fashion command pattern is used. Here methods get class status and executing class generic method actually calls different class method for which the command object represents to. There can be two types of pattern here, a) there is separate command class registering the methods of a subject class and executing its (subject class) methods once execute method

of command is clicked, b) command class itself subclasses serve different behaviors and program needs to mix code with abstract command class execute method and the actual execute will be called in the context of subclasses behaviors. Lower level staff in any department is command executes for command received through various higher officials.

First approach: Class methods register their methods with the command class and when execute method of command class is executed a callback to already registered method is called. For example a robot makes many movements front, back, left, right. A function executing these movements needs these operations stored in a list so that just executing the list one by one in a similar fashion movement can take place. The function does not know about the robot and its particular movements. This can happen when a robot action gets registered with a new class called common class and when client executes its execute method, through callback robot action take place.



Code Example

```

package command;
sub new {
my ($class,$ref,$action)=@_;
bless {OBJREF=>$ref,ACTION=>$action},$class;
}
sub execute {
my $ref=shift;
$ref->{ACTION}($ref->{OBJREF});
}
1;
<<command.pm>>
  
```

```

package robot;
sub new {
my $class=shift;
bless {},$class;
}
sub back {
print "robot::back\n";
}
sub front {
print "robot::front\n";
}
sub right {
print "robot::right\n";
}
sub left {
  
```

```

print "robot::left\n"
}
1;
<<robot.pm>>

package invoker;
sub new {
my $class=shift;
bless {},$class;
}
sub action {
my ($ref,$actionlist)=@_;
print "making actions on robot\n";
map{$_->execute}@$actionlist;
}
1;
<<invoker.pm>>

use strict;
use warnings;
use invoker;
use robot;
use command;
my $robotref=new robot;
new invoker->action([new command($robotref,\&robot::left),new command($robotref,\&robot::right),new command($robotref,\&robot::front),new command($robotref,\&robot::back)]);
<<main.pl>>
---data---
making actions on robot
robot::left
robot::right
robot::front
robot::back
-----

```

State

A class behavior may change when the state of a data type changes so class function does different operation depending upon the state of the data type. Behavior code is close to change when doing the operation based on state code where the state is separated from the main behavior and when required behavior passes the logic to a separate state abstraction. This kind of state abstraction is state pattern. This makes behavior and logic to be separated and logic is in state classes, making behavior close to changes and open for logic to be extended in the shape of more state subclasses possible to be added at run time. For example a lift can sustain 5 people. A person can only use the lift when there is less than 5 people I the lift. Lift behavior includes the opening and closing of lift doors, getting lift move up and down. Lift states include lift at rest, lift serving people up, lift serving people down, lift full On electrical switch board there is a plug point and a switch. When a user inserts pin of an electrical equipment in the plug, Bart passes this information to the switch. Switch internally maintains various states (i.e. On and off), one state at a time would be effective and effective state receives the information. It is a state which decides the action of the plug and typically on allows the connection whereas off doesn't allowing.

-----	+-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+	+-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+	+-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+
u	-> lift	<>----->	state
s	+-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+		+-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+
e	_state:state		_waitinstate:waitingstate
r	+-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+		_upstate:upstate
-----	mvflr():void		_downstate:downstate
	open():void		_fullupstate:fullupstate
	close():void		_fulldownstate:fulldownst
	nextflr():void		+-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+

```

+-----+
| / \
| -
|
+-----+
| liftevent |
+-----+
| enter():void |
| exit():void |
| prbbtn():void|
+-----+

+-----+
| open():void
| close():void
| enter():void
| exit():void
| mvflr():void
| prbbtn():void
| nextflr():void
+-----+

+-----+
| / \
| -
|
+-----+

+-----+
| | | | |
+-----+ +-----+ +-----+ +-----+ +-----+
|waitingstate| |upstate| |downstate| |fullupstate| |fulldownstate|
+-----+ +-----+ +-----+ +-----+ +-----+
| prbbtn():vo | | nextflr(| | nextflr(| | nextflr():| | nextflr():vo|
| id | | ):void | | ):void | | void | | id |
| close():voi | | close():| | close() | | close():vo | | close():void|
| d | | void | | :void | | id | +-----+
+-----+ +-----+ +-----+ +-----+

```

Code Example

```

package scheduler;
sub new {
my ($class,@item)=@_;
bless [@item],$class;
}
sub start {
my $ref=shift;
my $stop=0;
while(!$stop) {
foreach(@$ref) {
if($_->execute() eq "stop"){
$stop=1;
last;
}
}
sleep(1);
}
}
sub register {
my ($ref,@item)=@_;
push @$ref,@item;
}
sub unregister {
my ($ref,$item)=@_;
splice(@$ref,(grep($ref->[$_] eq $item,0..$#{$ref}))[0],1);
}
1;
<<scheduler.pm>>

package handler;
sub new {
my ($class,$ref)=@_;
bless $ref,$class;
}

```

```

sub execute {
my $ref=shift;
$ref->evaluate();
}
1;
<<handler.pm>>

package person;
use base qw(handler);
sub new {
my ($class,$scheduler,$lift)=@_;
$class->SUPER::new({users=>[],scheduler=>$scheduler,lift=>$lift,begintime=>time});
}
sub adduser {
my ($ref,@users)=@_;
push @{$ref->{users}},@users;
}
sub execute {
my $ref=shift;
if(! scalar @{$ref->{users}}) {
$ref->{scheduler}->unregister($ref);
} else {
my $i=0;
while(1) {
last if $i >= scalar @{$ref->{users}};
if($ref->{users}[$i]{entertime}<(time-$ref->{begintime})) {
$ref->{scheduler}->register($ref->{users}[$i]);
splice(@{$ref->{users}},$i,1);
}else {
$i=$i+1;
}
}
}
sub getlift {
my $ref=shift;
$ref->{lift};
}
1;
<<person.pm>>

package user;
use base qw(handler person);
sub new {
my ($class,$person,$entertime,$enterfloor,$exitfloor)=@_;
$class->SUPER::new({person=>$person,entertime=>$entertime,enterfloor=>$enterfloor,exitfloor=>
$exitfloor});
}
sub execute {
my $ref=shift;
my $canenter=0;
if(defined $ref->{entertime}) {
if($ref->{enterfloor}>$ref->{exitfloor}) {
$ref->{person}->getlift()->btn($ref->{enterfloor},2) if $ref->{person}->getlift()-
>isopen() !=$ref->{enterfloor};
}else {
$ref->{person}->getlift()->btn($ref->{enterfloor},1) if $ref->{person}->getlift()-
>isopen() !=$ref->{enterfloor};
}
}
}

```

```

$ref->{entertime}=undef;
}
if(!defined $ref->{entertime}) {
  if(defined $ref->{exitfloor} && !defined $ref->{enterfloor} && $ref->{person}->getlift()-
>isopen()==$ref->{exitfloor}) {
    $ref->{person}->getlift()->gtinout(1,$ref);
    $ref->{exitfloor}=undef;
  }else {
    if(defined $ref->{enterfloor} && $ref->{person}->getlift()->isopen()==$ref->{enterfloor}) {
      if($ref->{exitfloor}>$ref->{enterfloor}) {
        if(!($ref->{person}->getlift()->{BUTTON}[$ref->{enterfloor}]&1)) {
          $canenter=1;
        }
      }else {
        if(!($ref->{person}->getlift()->{BUTTON}[$ref->{enterfloor}]&2)) {
          $canenter=1;
        }
      }
    }
    if($canenter) {
      $ref->{person}->getlift()->gtinout(2,$ref);
      $ref->{person}->getlift()->btn($ref->{exitfloor},4);
      $ref->{enterfloor}=undef;
    }
  }
  1;
}<<user.pm>>

package lift;
use base qw(handler);
sub new {
  my ($class,$scheduler,$state)=@_;
  $class->SUPER::new({BUTTON=>[],FLOOR=>0,openstate=>0,liftmving=>0,PEOPLE=>0,MAXALLOWED=>5,
FLOOR=>5,nextflrwaittime=>0,closedrwaittime=>0,scheduler=>$scheduler,STATE=>$state});
}
sub open {
  my $ref=shift;
  print "door opened at floor:",$ref->{FLOOR},"\\n";
  $ref->{closedrwaittime}=time+1;
  $ref->{openstate}=1;
}
sub isopen() {
  my $ref=shift;
  if($ref->{openstate}) {
    $ref->{FLOOR};
  }else {
    $ref->{TOPFLOOR}+1;
  }
}
sub close {
  my $ref=shift;
  print "door closed at floor:",$ref->{FLOOR},"\\n";
  $ref->{openstate}=0;
  $ref->{STATE}->close($ref);
}
sub mvflr {
  my $ref=shift;

```

```

$ref->{nextflrwaittime}=time+2;
$ref->{liftmving}=1;
}
sub nextflr {
my $ref=shift;
$ref->{liftmving}=0;
$ref->{STATE}->nextflr($ref);
}
sub state {
my $ref=shift;
if(scalar @_ ) {
print "old state:",$ref->{STATE};
$ref->{STATE}=shift;
print ": changed to ",$ref->{STATE},"\\n";
}else {
$ref->{STATE};
}
}
sub scheduler {
$_[0]->{scheduler};
}
sub execute {
my $ref=shift;
$ref->{STATE}->execute($ref);
}
1;
<<lift.pm>>

package liftevent;
use base qw(lift);
sub btn {
my $ref=shift;
$ref->state->btn($ref,@_);
}
sub gtinout {
my $ref=shift;
$ref->state->gtinout($ref,@_);
}
sub bttnenable {
my $ref=shift;
$ref->state->bttnenable($ref,@_);
}
1;
<<liftevent.pm>>

package state;
use upstate;
use downstate;
use fullupstate;
use fulldownstate;
use waitingstate;
my ($waitingstate,$upstate,$downstate,$fullupstate,$fulldownstate)=undef;
sub waitingstate {
$waitingstate='waitingstate'->new if !defined $waitingstate;
$waitingstate;
}
sub upstate {
$upstate='upstate'->new if !defined $upstate;;
$upstate;

```

```

}
sub downstate {
$downstate='downstate'->new if !defined $downstate;
$downstate;
}
sub fullupstate {
$fullupstate='fullupstate'->new if !defined $fullupstate;
$fullupstate;
}
sub fulldownstate {
$fulldownstate='fulldownstate'->new if !defined $fulldownstate;
$fulldownstate;
}
sub new {
my $class=shift;
bless {},$class;
}
sub btn {
my ($ref,$lift,$targetfloor,$updowninside)=@_;
$lift->{BUTTON}[$targetfloor]=$updowninside;
}
sub gtinout {
my ($ref,$lift,$inout,$user)=@_;
if($inout==2) {
$lift->{PEOPLE}=$lift->{PEOPLE}+1;
print "one person getting in, PEOPLE:",$lift->{PEOPLE},"\\n";
}else {
$lift->{PEOPLE}=$lift->{PEOPLE}-1;
print "one person getting out, PEOPLE:",$lift->{PEOPLE},"\\n";
$lift->scheduler->unregister($user);
}
}
sub bttnenable {
my ($ref,$lift,$floor,$updown,$bttn)=@_;
my $count=0;
if($floor<0 && $floor>$lift->{TOPFLOOR}) {
0;
}else {
if($updown==1) {
foreach($floor..$lift->{TOPFLOOR}) {
$count=$count+($lift->{BUTTON}[$_]&$bttn);
}
}else {
foreach(0..$floor) {
$count=$count+($lift->{BUTTON}[$_]&$bttn);
}
}
}
$count;
}
sub execute {
my ($ref,$lift)=@_;
if($lift->{openstate} && time>$lift->{closedrwaittime}) {
$lift->close;
}else {
if($lift->{liftmving} && time>$lift->{nextflrwaittime}) {
$lift->nextflr;
}
}
}

```



```

}
1;
<<state.pm>>

package waitingstate;
use base qw(state);
sub btn {
my ($ref,$lift,$floor,$btn)=@_;
$ref->SUPER::btn($lift,$floor,$btn);
if($lift->{BUTTON}[$lift->{FLOOR}] & 7) {
$lift->{BUTTON}[$lift->{FLOOR}]=0;
$lift->open;
}
}
sub close() {
my ($ref,$lift)=@_;
if($lift->btnenable($lift->{FLOOR}+1,1,4)) {
$lift->state($ref->upstate);
}elseif($lift->btnenable($lift->{FLOOR}-1,2,4)) {
$lift->state($ref->downstate);
}elseif($lift->btnenable($lift->{FLOOR}+1,1,3)) {
$lift->state($ref->upstate);
}elseif($lift->btnenable($lift->{FLOOR}-1,2,3)) {
$lift->state($ref->downstate);
}

if($lift->{PEOPLE}>=$lift->{MAXALLOWED}) {
if($lift->state==$ref->upstate) {
$lift->state($ref->fullupstate);
}else {
if($lift->state==$ref->downstate) {
$lift->state=$ref->fulldownstate;
}
}
}

if($lift->state != $ref->waitingstate) {
$lift->mvflr;
}
}
1;
<<waitingstate.pm>>

package upstate;
use base qw(state);
sub nextflr {
my ($ref,$lift)=@_;
$lift->{FLOOR}=$lift->{FLOOR}+1;
print "upstate:next floor:",$lift->{FLOOR},"\\n";
if($lift->{BUTTON}[$lift->{FLOOR}]&5) {
$lift->{BUTTON}[$lift->{FLOOR}]&=2;
$lift->open;
}else {
$ref->close($lift);
}
}
sub close {
my ($ref,$lift)=@_;
if(!$lift->btnenable($lift->{FLOOR}+1,1,7)) {

```

```

if($lift->bttnenable($lift->{FLOOR},2,7)) {
$lift->state($ref->downstate);
if($lift->{BUTTON}[$lift->{FLOOR}]&2) {
$lift->{BUTTON}[$lift->{FLOOR}]&=5;
$lift->open;
return;
}
}else {
$lift->state($ref->waitingstate);
}
}
if($lift->{PEOPLE}>=$lift->{MAXALLOWED}) {
if($lift->state == $ref->upstate) {
$lift->state($ref->fullupstate);
}else {
if($lift->state == $ref->downstate) {
$lift->state($ref->fulldownstate);
}
}
}
if($lift->state != $ref->waitingstate) {
$lift->mvflr;
}
}
1;
<<upstate.pm>>

package downstate;
use base qw(state);
sub nextflr {
my ($ref,$lift)=@_;
$lift->{FLOOR}=$lift->{FLOOR}-1;
print "downstate:next floor:",$lift->{FLOOR},"\\n";
if($lift->{BUTTON}[$lift->{FLOOR}]&6) {
$lift->{BUTTON}[$lift->{FLOOR}]&=1;
$lift->open;
}else {
$ref->close($lift);
}
}
sub close {
my ($ref,$lift)=@_;
if(!$lift->bttnenable($lift->{FLOOR}-1,2,7)) {
if($lift->bttnenable($lift->{FLOOR},1,7)) {
$lift->state($ref->upstate);
if($lift->{BUTTON}[$lift->{FLOOR}]&1) {
$lift->open;
$lift->{BUTTON}[$lift->{FLOOR}]&=6;
return;
}
}else {
$lift->state($ref->waitingstate);
}
}
if($lift->{PEOPLE}>=$lift->{MAXALLOWED}) {
if($lift->state == $ref->downstate) {
$lift->state($ref->fulldownstate);
}elseif($lift->state==$ref->fullup) {
$lift->state($ref->fullupstate);
}
}
}

```

```

}
}
if($lift->state!=$ref->waitingstate) {
$lift->mvflr;
}
}
1;
<<downstate.pm>>

package fullupstate;
use base qw(state);
sub nextflr {
my ($ref,$lift)=@_;
$lift->{FLOOR}=$lift->{FLOOR}+1;
print "fullupstate:next floor:",$lift->{FLOOR},"\\n";
if($lift->{BUTTON}[$lift->{FLOOR}]&4) {
$lift->{BUTTON}[$lift->{FLOOR}]&3;
$lift->open;
}else {
$ref->close($lift);
}
}
sub close {
my ($ref,$lift)=@_;
if($lift->{PEOPLE}<=$lift->{MAXCOUNT}) {
$lift->state($ref->upstate);
$lift->state->close($lift);
}else {
if(!$lift->bttnenable($lift->{FLOOR}+1,1,4)) {
if($lift->bttnenable($lift->{FLOOR}-1,2,4)) {
$lift->state($ref->fulldownstate);
}else {
$lift->state($ref->waitingstate);
}
}
if($lift->state!=$ref->waitingstate) {
$lift->mvflr;
}
}
}
1;
<<fullupstate.pm>>

package fulldownstate;
use base qw(state);
sub nextflr {
my ($ref,$lift)=@_;
$lift->{FLOOR}=$lift->{FLOOR}-1;
print "fulldownstate:next floor:",$lift->{FLOOR},"\\n";
if($lift->{BUTTON}[$lift->{FLOOR}]&4) {
$lift->{BUTTON}[$lift->{FLOOR}]&3;
$lift->open;
}else {
$ref->close($lift);
}
}
sub close {
my ($ref,$lift)=@_;
if($lift->{PEOPLE}<$lift->{MAXALLOWED}) {

```

```

$lift->state($ref->downstate);
$lift->state->close($lift);
}else {
if(!$lift->bttnenable($lift->{FLOOR}-1,2,4)) {
if($lift->bttnenable($lift->{FLOOR}+1,1,4)) {
$lift->state($ref->fullupstate);
}else {
$lift->state($ref->waitingstate);
}
}
if($lift->state!=$ref->waitingstate) {
$lift->mvflr;
}
}
}
1;
<<fulldownstate.pm>>

use strict;
use warnings;

use scheduler;
use person;
use user;
use liftevent;
use waitingstate;
$|=1;
my $scheduler=new scheduler;
my $lift=new liftevent($scheduler,'waitingstate'->waitingstate);
my $person=new person($scheduler,$lift);
$person->adduser(new user($person,0,0,1),new user($person,0,5,0),new user($person,0,1,5),new
user($person,10,0,3),new user($person,4,5,2),new user($person,5,2,3),new user($person,5,4,1),new
user($person,3,0,2),new user($person,14,3,0),new user($person,2,3,4),new user($person,0,2,5),new
user($person,0,4,0),new user($person,0,3,1),new user($person,0,0,1));
$scheduler->register($person,$lift);
$scheduler->start;
<<main.pl>>
---data---
door opened at floor:0
one person getting in, PEOPLE:1
one person getting in, PEOPLE:2
door closed at floor:0
old state:waitingstate=HASH(0x78b044): changed to upstate=HASH(0x52fd54)
upstate:next floor:1
door opened at floor:1
one person getting out, PEOPLE:1
one person getting in, PEOPLE:2
one person getting out, PEOPLE:1
door closed at floor:1
upstate:next floor:2
door opened at floor:2
one person getting in, PEOPLE:2
one person getting in, PEOPLE:3
door closed at floor:2
upstate:next floor:3
door opened at floor:3
one person getting in, PEOPLE:4
one person getting out, PEOPLE:3
door closed at floor:3

```

```
upstate:next floor:4
door opened at floor:4
one person getting out, PEOPLE:2
door closed at floor:4
upstate:next floor:5
door opened at floor:5
one person getting out, PEOPLE:1
one person getting out, PEOPLE:0
door closed at floor:5
old state:upstate=HASH(0x52fd54): changed to downstate=HASH(0x52ff64)
door opened at floor:5
one person getting in, PEOPLE:1
one person getting in, PEOPLE:2
door closed at floor:5
downstate:next floor:4
door opened at floor:4
one person getting in, PEOPLE:3
one person getting in, PEOPLE:4
door closed at floor:4
downstate:next floor:3
door opened at floor:3
one person getting in, PEOPLE:5
one person getting in, PEOPLE:6
door closed at floor:3
old state:downstate=HASH(0x52ff64): changed to fulldownstate=HASH(0x52fc24)
fulldownstate:next floor:2
door opened at floor:2
one person getting out, PEOPLE:5
door closed at floor:2
fulldownstate:next floor:1
door opened at floor:1
one person getting out, PEOPLE:4
one person getting out, PEOPLE:3
door closed at floor:1
old state:fulldownstate=HASH(0x52fc24): changed to downstate=HASH(0x52ff64)
downstate:next floor:0
door opened at floor:0
one person getting out, PEOPLE:2
one person getting out, PEOPLE:1
one person getting out, PEOPLE:0
door closed at floor:0
old state:downstate=HASH(0x52ff64): changed to upstate=HASH(0x52fd54)
door opened at floor:0
one person getting in, PEOPLE:1
one person getting in, PEOPLE:2
door closed at floor:0
upstate:next floor:1
door opened at floor:1
door closed at floor:1
upstate:next floor:2
door opened at floor:2
one person getting out, PEOPLE:1
door closed at floor:2
upstate:next floor:3
door opened at floor:3
one person getting out, PEOPLE:0
door closed at floor:3
old state:upstate=HASH(0x52fd54): changed to waitingstate=HASH(0x78b044)
-----
```

State pattern way of solving arithmetic equations:



Code Example

```

package calculator;
use result;
use addition;
use division;
use multiplication;
use subtraction;
sub new {
my ($class,$state)=@_;
bless {STATE=>$state},$class;
}
sub setstate {
my ($ref,$state)=@_;
$ref->{STATE}=$state;
}
sub calculate {
my ($ref,$expression)=@_;
my $state=$ref->{STATE};
$ref->{STATE}->calculate($expression,$ref);
while($state != $ref->{STATE}) {
$expression=$ref->{STATE}->calculate($expression,$ref);
}
$expression;
}
sub printonscreen {
my ($ref,$string)=@_;
print $string;
}

```

```

1;
<<calculator.pm>>

package state;
my ($addition,$subtraction,$multiplication,$division,$result)=undef;
sub additionstate {
    $addition=new('addition') if !defined $addition;
    $addition;
}
sub subtractionstate {
    $subtraction=new('subtraction') if !defined $subtraction;
    $subtraction;
}
sub multiplicationstate {
    $multiplication=new('multiplication') if !defined $multiplication;
    $multiplication;
}
sub divisionstate {
    $division=new('division') if !defined $division;
    $division;
}
sub result {
    $result=new('result') if !defined $result;
    $result;
}
sub new {
    my $class=shift;
    bless {},$class;
}

sub calculate {
    my ($ref,$expression,$operator,$calculatorref)=@_;
    while($expression=~/(-?\d+)(\d+)(-?\d+)/) {
        $calculatorref->prontonscreen("$1:$3\n");
        $expression=join('',$` ,eval "$1$2$3",$`);
    }
    $expression;
}
1;
<<state.pm>>

package addition;
use base qw(state);
sub calculate {
    my ($ref,$expression,$calculatorref)=@_;
    $calculatorref->prontonscreen("adding...:expression:$expression\n");
    $expression=__PACKAGE__->SUPER::calculate($expression,'\+', $calculatorref);
    $calculatorref->setstate(__PACKAGE__->SUPER::result);
    $expression;
}
1;
<<addition.pm>>

package subtraction;
use base qw(state);
sub calculate {
    my ($ref,$expression,$calculatorref)=@_;
    $calculatorref->prontonscreen("subtracting...:expression:$expression\n");
    $expression=__PACKAGE__->SUPER::calculate($expression,'-', $calculatorref);

```

```

$calculatorref->setstate(__PACKAGE__->SUPER::additionstate);
$expression;
}
1;
<<subtraction.pm>>

package multiplication;
use base qw(state);
sub calculate {
my ($ref,$expression,$calculatorref)=@_;
$calculatorref->printonscreen("multiplying...:expression:$expression\n");
$expression=__PACKAGE__->SUPER::calculate($expression,'*', $calculatorref);
$calculatorref->setstate(__PACKAGE__->SUPER::subtractionstate);
$expression;
}
1;
<<multiplication.pm>>

package division;
use base qw(state);
sub calculate {
my ($ref,$expression,$calculatorref)=@_;
$calculatorref->printonscreen("division...");
SUPER::calculate($expression,'[ / ]');
$calculatorref->setstate->(SUPER::multiplication);
$expression;
}
1;
package division;
use base qw(state);
sub calculate {
my ($ref,$expression,$calculatorref)=@_;
$calculatorref->printonscreen("dividing...:expression:$expression\n");
$expression=__PACKAGE__->SUPER::calculate($expression,'/', $calculatorref);
$calculatorref->setstate(__PACKAGE__->SUPER::multiplicationstate);
$expression;
}
1;
<<division.pm>>

package result;
use base qw(state);
sub calculate {
my ($ref,$expression,$calculatorref)=@_;
$calculatorref->setstate($ref->SUPER::divisionstate());
$expression;
}
1;
<<result.pm>>

use strict;
use warnings;
use calculator;
use state;

print "result:",new calculator(state::result)->calculate('1+3/3*2-2+6/2/3-2'),"\n";
print "result:",new calculator(state::result)->calculate('1-2+4*3-6/2+8*3-2*70/10');
<<main.pl>>

```



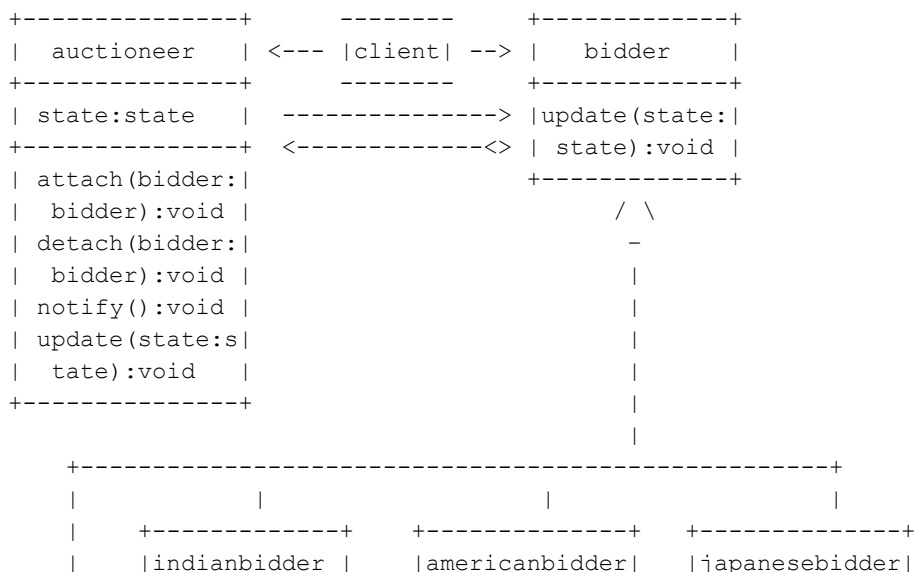
```

---data---
dividing...:expression:1+3/3*2-2+6/2/3-2
3:3
6:2
3:3
multiplying...:expression:1+1*2-2+1-2
1:2
subtracting...:expression:1+2-2+1-2
2:2
1:2
adding...:expression:1+0+-1
1:0
1:-1
result:0
dividing...:expression:1-2+4*3-6/2+8*3-2*70/10
-6:2
70:10
multiplying...:expression:1-2+4*3-3+8*3-2*7
4:3
8:3
-2:7
subtracting...:expression:1-2+12-3+24-14
1:2
12:3
24:14
adding...:expression:-1+9+10
-1:9
8:10
result:18
-----

```

Observer

When a subject has to be observed by many observers, leading to a data view/controller model, makes an observer pattern. A subject which keeps the data lets observers registered with it, data changes in the subject trigger event sent to all the observers who can update their status. One of the observer may again make some tuning in the subject leading to data change in the subject which make other observers get informed For example, in an auction an auctioneer is a subject where as bidders are observers. Auctioneer starts with an initial value and let the bidders (observers/view) hear it. An observer (controller) makes a bid and subject (model/data) changes the auction rate and announces again it at all the observers (viewers). Typically, any general election happens and many observers get placed there from many different countries.



```

|      +-----+      +-----+      +-----+
|      | state:state |      | state:state |      | state:state |
|      +-----+      +-----+      +-----+
|      | update(state|      | update(state:|      | update(state:|
|      | :state):void|      | state):void |      | state):void |
|      +-----+      +-----+      +-----+
|
+-----+
|      |      |      |
+-----+ +-----+ +-----+
|canadianbidder| |austrailianbidder| |chinesebidder|
+-----+ +-----+ +-----+
| state:state | | state:state | | state:state |
+-----+ +-----+ +-----+
| update(state:| | update(state:sta| | update(state|
| state):void | | ate):void | | :state)void|
+-----+ +-----+ +-----+

```

Code Example

```

package scheduler;
sub new {
my ($class,@item)=@_;
bless [@item],$class;
}
sub start {
my $ref=shift;
my $stop=0;
while(!$stop) {
foreach (@$ref) {
if($_->execute() eq "stop"){
$stop=1;
last;
}
}
}
}
sub register {
my ($ref,$item)=@_;
push @$ref,$item;
}
sub unregister {
my ($ref,$item)=@_;
delete @$ref[grep($$ref[$_] =~ /$item/, 0..$#$ref)];
}
1;
<<scheduler.pm>>
package sharedmem;
my $instance=undef;
sub instance {
my $class=shift;
if(!defined $instance) {
$instance=__PACKAGE__->new();
}else {
$instance;
}
}
sub new {
my $class=shift;

```

```

bless [], $class;
}
sub getfromindex {
my ($ref, $index) = @_;
$ref->[$index];
}
sub setinindex {
my ($ref, $index, $val) = @_;
$ref->[$index] = $val;
}
1;
<sharedmem.pm>>

package handler;
sub new {
my ($class, $ref) = @_;
bless $ref, $class;
}
sub execute {
my $ref = shift;
$ref->evaluate();
}
1;
<<handler.pm>>

package subject;
use base qw(handler);
sub new {
my ($class, $value, $adjustfactor, $sharedmem) = @_;
$class->SUPER::new({value => $value, adjustfactor => $adjustfactor, sharedmem => $sharedmem, announce =
> 0, observers => []});
}
sub register {
my ($ref, @observer) = @_;
push @{$ref->{observers}}, @observer;
}
sub evaluate {
my $ref = shift;
my $maximum = $ref->{value};
foreach (split(':', $ref->{sharedmem}->getfromindex(2))) {
if ($_ > $maximum) {
$maximum = $_;
}
}
$ref->{sharedmem}->setinindex(2, undef);
if ($ref->{value} == $maximum) {
if ($ref->{announce} == 3) {
print "all three announcement over, final bidding at:", $ref->{value}, "\n";
return "stop";
} else {
$ref->{announce} = $ref->{announce} + 1;
print "making announcement number:", $ref->{announce}, ", at:", $ref->{value}, "\n";
$ref->{sharedmem}->setinindex(1, $ref->{sharedmem}->getfromindex(1) + $ref->{adjustfactor});
}
} else {
$ref->{announce} = 0;
$ref->{sharedmem}->setinindex(1, 0);
$ref->{value} = $maximum;
print "auctioneer: new bidding rate:", $maximum, "\n";

```

```

}
map ($_->notify(), @{$ref->{observers}});
$ref->{sharedmem}->setinindex(0,$ref->{value});
#$ref->{sharedmem}->setinindex(1,$ref->{adjustfactor});
}
1;
<<subject.pm>>

package observer;
use base qw(handler);
sub new {
my ($class,$bidvalue,$sharedmem)=@_;
$class->SUPER::new({bidvalue=>$bidvalue,sharedmem=>$sharedmem});
}
sub notify {
my $ref=shift;
$ref->{notified}=1;
}
sub bid {
my $ref=shift;
print "bidder:bidding at:",$ref->{bidvalue},"\\n";
$ref->{sharedmem}->setinindex(2,join(':', $ref->{sharedmem}->getfromindex(2), $ref->{bidvalue}));
}
sub evaluate {
my $ref=shift;
if($ref->{notified}) {
$ref->{notified}=0;
if($ref->{bidvalue}>$ref->{sharedmem}->getfromindex(0) && $ref->{bidvalue}<($ref->{sharedmem}->getfromindex(0)+$ref->{sharedmem}->getfromindex(0)*$ref->{sharedmem}->getfromindex(1)/100)) {
$ref->bid;
}
}
}
1;
<<observer.pm>>

use strict;
use warnings;

use scheduler;
use subject;
use observer;
use sharedmem;

my $sharedmem=new sharedmem;
my $subject=new subject(10,110,$sharedmem);
my $indianbidder=new observer(100,$sharedmem);
my $canadianbidder=new observer(40,$sharedmem);
my $americanbidder=new observer(80,$sharedmem);
my $chinesebidder=new observer(90,$sharedmem);
my $austrailianbidder=new observer(10,$sharedmem);
my $japanesebidder=new observer(120,$sharedmem);
$subject->register($indianbidder,$canadianbidder,$americanbidder,$chinesebidder,$austrailianbidder,$japanesebidder);
new scheduler($subject,$indianbidder,$canadianbidder,$americanbidder,$chinesebidder,$austrailianbidder,$japanesebidder)->start;
<<main.pl>>

```

```

---data---
making announcement number:1, at:10
making announcement number:2, at:10
making announcement number:3, at:10
bidder:bitding at:40
auctioneer:new bitding rate:40
making announcement number:1, at:40
bidder:bitding at:80
auctioneer:new bitding rate:80
making announcement number:1, at:80
bidder:bitding at:100
bidder:bitding at:90
bidder:bitding at:120
auctioneer:new bitding rate:120
making announcement number:1, at:120
making announcement number:2, at:120
making announcement number:3, at:120
all three announcement over, final bitding at:120
-----

```

Arithmetic expression solution with observer:

```

+-----+
| subject | <-----|client| -----
+-----+ <-----| ----- |> | observer |
| register(observer):void | -----| | -----
| notify():void | | |-----<> | update(express|
| update(state:st| |-----> | ion:string):v|
| ate):void | | oid |
+-----+
|
| / \
| -
|
+-----+
| | | |
+-----+ +-----+ +-----+ +-----+
| division | |multiplication | | subtraction | | addition |
+-----+ +-----+ +-----+ +-----+
| update(express| | update(express| | update(express| | update(expre|
| ion:string):v| | ion:string):v| | sion:string)| | ssion:stin|
| oid | | oid | | void | | g):void |
+-----+ +-----+ +-----+ +-----+

```

Code Example

```

package scheduler;
sub new {
my ($class,@item)=@_;
bless [@item],$class;
}
sub start {
my $ref=shift;
my $stop=0;
while(!$stop) {
foreach (@$ref) {
if($_->execute() eq "stop"){
$stop=1;
last;

```

```

}
}
}
}
sub register {
my ($ref,$item)=@_;
push @$ref,$item;
}
sub unregister {
my ($ref,$item)=@_;
delete @$ref[grep($$ref[$_] =~ /$item/, 0..$#$ref)];
}
1;
<<scheduler.pm>>
package sharedmem;
my $instance=undef;
sub instance {
my $class=shift;
if(!defined $instance) {
$instance=__PACKAGE__->new();
}else {
$instance;
}
}
sub new {
my $class=shift;
bless [],$class;
}
sub getfromindex {
my ($ref,$index)=@_;
$ref->[$index];
}
sub setinindex {
my ($ref,$index,$val)=@_;
$ref->[$index]=$val;
}
1;
<sharedmem.pm>>

package handler;
sub new {
my $class=shift;
bless {},$class;
}
sub execute {
my $ref=shift;
$ref->evaluate();
}
1;
<<handler.pm>>

package observer;
use sharedmem;
use base qw(handler);
sub evaluate {
my $ref=shift;
my $expression;
if($ref->{notified}) {
$expression='sharedmem'->instance->getfromindex(0);

```

```

while ($expression=~/(-?\d+)($ref->{operator})(-?\d+)/) {
$expression=join('',$`,eval "$1$2$3",$');
}
'sharedmem'->instance->setinindex(2,$expression);
'sharedmem'->instance->setinindex(1,0);
$ref->{notified}=false;
}
}
sub notify {
my $ref=shift;
$ref->{notified}=true;
}
sub getexpression {
'sharedmem'->instance->getfromindex(0);
}
1;
<<observer.pm>>

package divider;
use base qw(observer);
sub new {
my $ref=__PACKAGE__->SUPER::new;
$ref->{operator}='/';
$ref;
}
sub evaluate {
my $ref=shift;
if($ref->SUPER::getexpression()=~/\d$ref->{operator}\d/) {
print "divider:",$ref->SUPER::getexpression(),"\n";
$ref->SUPER::evaluate($ref->SUPER::getexpression());
}
}
1;
<<divider.pm>>

package subtracter;
use base qw(observer);
sub new {
my $class=shift;
my $ref=__PACKAGE__->SUPER::new;
$ref->{operator}='-';
$ref;
}
sub evaluate {
my $ref=shift;
if($ref->SUPER::getexpression()!~/\d[*\/]\d/ && $ref->SUPER::getexpression()=~/\d$ref->{operator}\d/) {
printf "subtracter:%s\n",$ref->SUPER::getexpression();
$ref->SUPER::evaluate($ref->SUPER::getexpression());
}
}
1;
<<subtracter.pm>>

package multiplier;
use base qw(observer);
sub new {
my $ref=__PACKAGE__->SUPER::new;
$ref->{operator}='\*';

```

```

$ref;
}
sub evaluate {
my $ref=shift;
if($ref->SUPER::getexpression()!~/\// && $ref->SUPER::getexpression()=~/\d$ref-
>{operator}\d/) {
print "multiplier:",$ref->SUPER::getexpression(),"\n";
$ref->SUPER::evaluate($ref->SUPER::getexpression());
}
}
1;
<<multiplier.pm>>

package adder;
use base qw(observer);
sub new {
my $ref=__PACKAGE__->SUPER::new;
$ref->{operator}='\+';
$ref;
}
sub evaluate {
my $ref=shift;
if($ref->SUPER::getexpression()!~/\d[-*\/]\d/ && $ref->SUPER::getexpression()=~/\d$ref-
>{operator}\d/) {
print "adder:",$ref->SUPER::getexpression(),"\n";
$ref->SUPER::evaluate();
}
}
1;
<<adder.pm>>

package subject;
use sharedmem;
use base qw(handler);
sub new {
my ($class,$expression)=@_;
'sharedmem'->instance->setinindex(2,$expression);
'sharedmem'->instance->setinindex(1,0);
bless [],$class;
}
sub execute {
my $ref=shift;
if('sharedmem'->instance->getfromindex(1)==1) {
print "result:",'sharedmem'->instance->getfromindex(2);
"stop";
}else {
'sharedmem'->instance->setinindex(1,1);
'sharedmem'->instance->setinindex(0,'sharedmem'->instance->getfromindex(2));
map {$_->notify} @$ref;
}
}
sub register {
my ($ref,@item)=@_;
push @$ref,@item;
}
1;
<<subject.pm>>

use strict;

```



```

use warnings;

use adder;
use multiplier;
use divider;
use subtracter;
use subject;
use scheduler;

my $adderref=new adder;
my $multiplierref=new multiplier;
my $dividerref=new divider;
my $subtracterref=new subtracter;

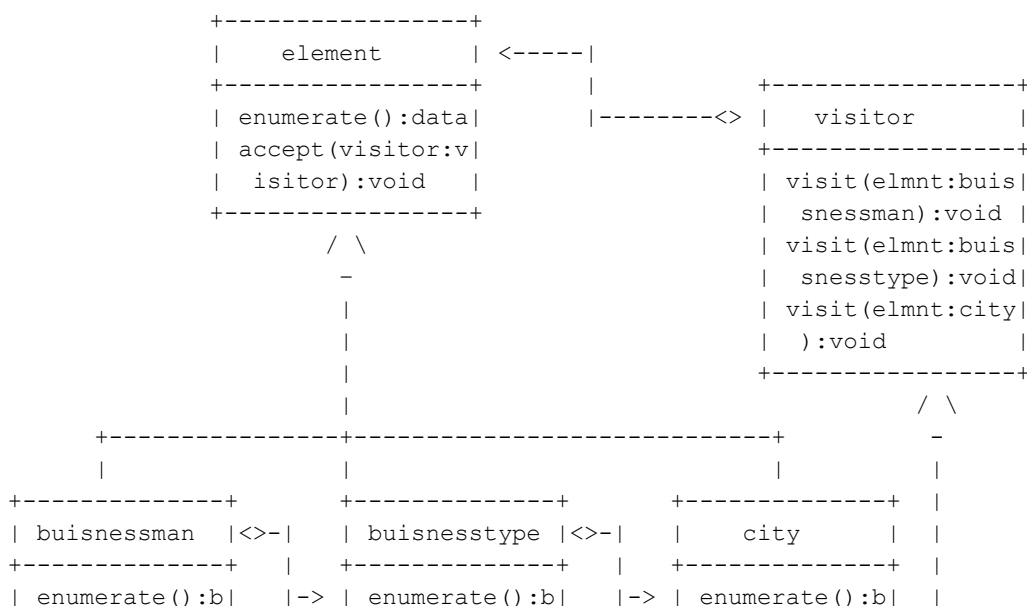
my $subjectref=new subject("1-2+4*3-6/2+8*3-2*70/10");
$subjectref->register($adderref,$multiplierref,$dividerref,$subtracterref);
new scheduler($subjectref,$adderref,$multiplierref,$dividerref,$subtracterref)->start;
<<main.pl>>

---data---
divider:1-2+4*3-6/2+8*3-2*70/10
multiplier:1-2+4*3-3+8*3-2*7
subtracter:1-2+12-3+24-14
adder:-1+9+10
result:18
-----

```

Visitor

A collection data type generally supports similar kind of operations. If a new operation is to be supported on a collection it would violate design principle, if collection class adds new operations and then it would be required at each collection class. Better solution is if a Separate dedicated interface does this. Each subclass of an interface adds a new operation this makes adding operation to a class at run time. This new interface is the visitor pattern as it visits collection class in order to provide new operation. For example, a data container abstract class provides data sorting operation, but does not provide list size, complexity calculation operation. A visitor class can have subclasses statistics, which calculate size and complexities of the member data respectively. Collection class can have a method which accepts these visitor classes and calling method actually passes the call to the visitor so that specific operation in the visitor implementation can take place. During a course of time, many visitors visit any province and write about its social, economic and political status. They collect this information from the province's resources only.



uisnesstype	uisnesstype	uisnesstype	
accept(vstr:v	accept(vstr:v	accept(vstr:v	
isitor):void	isitor):void	isitor):void	
+-----+	+-----+	+-----+	
			+-----+
			statisticsvisitor
			+-----+
			visit(elmnt:buisn
			essman):void
			visit(elmnt:buisn
			esstype):void
			visit(elmnt:city)
			:void
			+-----+

Code Example

```
package element;
sub new {
my ($class,$ref)=@_;
bless $ref,$class;
}
1;
<<element.pm>>

package buisnessman;
use base qw(element);
sub new {
my ($class,$name,@buisnesstype)=@_;
$class->SUPER::new({name=>$name,buisnesstype=>\@buisnesstype});
}
sub enumerate {
my $ref=shift;
$ref->{buisnesstype};
}
sub accept {
my ($ref,$visitor)=@_;
$visitor->visit($ref);
}
sub name {
$_[0]->{name};
}
1;
<<buisnessman.pm>>

package buisnesstype;
use base qw(element);
sub new {
my ($class,$name,@cities)=@_;
$class->SUPER::new({name=>$name,cities=>\@cities});
}
sub enumerate {
my $ref=shift;
$ref->{cities};
}
sub name {
$_[0]->{name};
}
1;
<<buisnesstype.pm>>
```

```

sub accept {
my ($ref,$visitor)=@_;
$visitor->visit($ref);
}
1;
<<buisnesstype.pm>>

package city;
use base qw(element);
sub new {
my ($class,$name)=@_;
$class->SUPER::new({name=>$name});
}
sub enumerate {
}
sub accept {
my ($ref,$visitor)=@_;
$visitor->visit($ref);
}
sub name {
$_[0]->{name};
}
1;
<<city.pm>>

package visitor;
sub new {
my $class=shift;
bless {},$class;
}
1;
<<visitor.pm>>

package statistics;
use base qw(visitor);
sub visit {
my ($ref,$element)=@_;
print $element->name,"\n";
map($_->accept($ref),@{$element->enumerate});
}
1;
<<statistics.pm>>

use strict;
use warnings;

use city;
use buisnesstype;
use businessman;
use statistics;
new businessman("person : tim",new buisnesstype("buisnesstype : hardware",new city("city :
mangalore"),new city("city : mysore"),new city("city : mandya")),new buisnesstype("buisnesstype
: software",new city("city : chennai"),new city("city : chaibasa"),new city("city : churu")),new
buisnesstype("buisnesstype : realstate",new city("city : delhi"),new city("city : dhaka"),new
city("city : dumka"))->accept(new statistics);
<<main.pl>>

---data---
```

```

person : tim
buisnesstype : hardware
city : mangalore
city : mysore
city : mandya
buisnesstype : software
city : chennai
city : chaibasa
city : churu
buisnesstype : realstate
city : delhi
city : dhaka
city : dumka
-----

```

Arithmetic expression solution through this pattern:

```

+-----+
| element | <----- |client| ----
+-----+ <----- +-----+
| execute(expressi | +> | visitor |
| on:string):void | |-----<> +-----+
| accpet(vstr:visi | | visitor(elemnt:|
| tor):void | | element):void|
+-----+ +-----+
/ \ / \
- -
| |
+-----+ |-----+
| | | |
+-----+ +-----+ +-----+ +-----+ |
|division|<>-| |multiplication|<>-| |subtract|<>-| |addition| |
+-----+ | +-----+ | +-----+ | +-----+ |
| accept(| |->| accept(vstr:v | |-> | accept(| |-> | accept(| |
| vstr:v | | visitor):void| | vstr:v | | vstr:vi | |
| isitor | +-----+ | visito | | sitor) | |
| ):void | | | ):void | | :void | |
+-----+ +-----+ +-----+ +-----+ |
| |
| statisticsvisitor |
+-----+
| visitor(elemnt:div |
| ision):void |
| visitor(elemnt:mul |
| tiplication):void |
| visitor(elmnt:subt |
| raction):void |
| visitor(elmnt:addi |
| tion):void |
+-----+

```

Code Example

```
package basedec;
sub new {
my $class=shift;
bless {ID=>$class},$class;
}
sub id {
my $ref=shift;
$ref->{ID};
}
sub execute {
my ($ref,$expression)=@_;
print $ref->id,":executing expression:",$expression->getexpression,"\n";
my $expressionstring=$expression->getexpression;
while($expressionstring=~/(?:\d+)(?:$ref->{operator})(?:\d+)/) {
$expressionstring=join('',$`,eval "$1$2$3",$');
}
$expression->setexpression($expressionstring);
}
sub accept {
my ($ref,$visitor,$expression)=@_;
$visitor->visit($ref,$expression);
}
1;
<<basedec.pm>>
```

```
package expression;
use base qw(basedec);
sub new {
my ($class,$expressionstring)=@_;
my $ref=__PACKAGE__->SUPER::new;
$ref->{expressionstring}=$expressionstring;
$ref;
}
sub getexpression {
my $ref=shift;
$ref->{expressionstring};
}
sub setexpression {
my ($ref,$expressionstring)=@_;
$ref->{expressionstring}=$expressionstring;
}
1;
<<expression.pm>>
```

```
package adder;
use base qw(basedec);
sub new {
my ($class,$decoratee)=@_;
my $ref=__PACKAGE__->SUPER::new;
$ref->{DECORATEE}=undef;
$ref->{operator}='+';
$ref;
}
sub execute {
my ($ref,$expression)=@_;
$ref->SUPER::execute($expression);
```

```
print "result:", $expression->getexpression, "\n";
}
1;
<<adder.pm>>
```

```
package divider;
use base qw(basedec);
sub new {
my ($class, $decoratee) = @_ ;
my $ref = __PACKAGE__->SUPER::new;
$ref->{DECORATEE} = $decoratee;
$ref->{operator} = '/';
$ref;
}
1;
<<divider.pm>>
```

```
package multiplier;
use base qw(basedec);
sub new {
my ($class, $decoratee) = @_ ;
my $ref = __PACKAGE__->SUPER::new;
$ref->{DECORATEE} = $decoratee;
$ref->{operator} = '*';
$ref;
}
1;
<<multiplier.pm>>
```

```
package subtracter;
use base qw(basedec);
sub new {
my ($class, $decoratee) = @_ ;
my $ref = __PACKAGE__->SUPER::new;
$ref->{DECORATEE} = $decoratee;
$ref->{operator} = '-';
$ref;
}
1;
<<subtracter.pm>>
```

```
package visitor;
sub new {
my $class = shift;
bless {}, $class;
}
1;
<<visitor.pm>>
```

```
package statisticsvisitor;
use base qw(visitor);
sub visit {
my ($ref, $component, $expression) = @_ ;
print "visitor, visiting:", $component->id, "\n";
$component->execute($expression);
$component->{DECORATEE}->accept($ref, $expression) if defined $component->{DECORATEE};
}
1;
<<statisticsvisitor.pm>>
```

```

use strict;
use warnings;
use basedec;
use visitor;
use statisticsvisitor;
use expression;
use adder;
use multiplier;
use divider;
use subtracter;

new divider(new multiplier(new subtracter(new adder)))->accept(new statisticsvisitor,new
expression('1+3/3*2-2+6/2/3-2'));
new divider(new multiplier(new subtracter(new adder)))->accept(new statisticsvisitor,new
expression('1-2+4*3-6/2+8*3-2*70/10'));
<<main.pl>>

---data---
visitor,visiting:divider
divider:executing expression:1+3/3*2-2+6/2/3-2
visitor,visiting:multiplier
multiplier:executing expression:1+1*2-2+1-2
visitor,visiting:subtracter
subtracter:executing expression:1+2-2+1-2
visitor,visiting:adder
adder:executing expression:1+0+-1
result:0
visitor,visiting:divider
divider:executing expression:1-2+4*3-6/2+8*3-2*70/10
visitor,visiting:multiplier
multiplier:executing expression:1-2+4*3-3+8*3-2*7
visitor,visiting:subtracter
subtracter:executing expression:1-2+12-3+24-14
visitor,visiting:adder
adder:executing expression:-1+9+10
result:18
-----

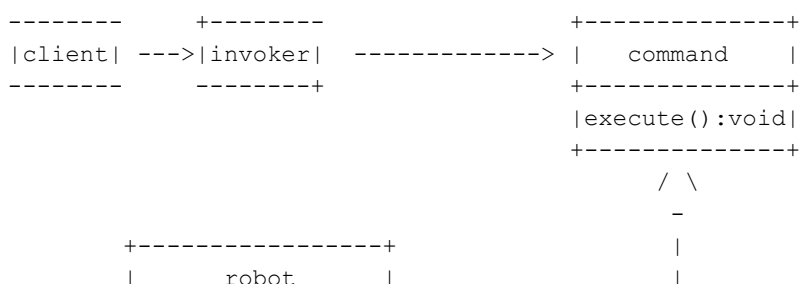
```

Direct Call Patterns

As per the name a method calls other directly and in non-recursive way and in non callback way.

Command (subclassed)

In this approach action taken by subject class is given separate object identity. Each action (command) subclass are actually calling the subject's respective action. The program needs to mix their code with the command abstract class with an execute method called against it and depending upon the subclass object actual action would take place.



```

+-----+
+----> | moveup():void |
|+---> | moveright():void| | |
||+--> | movefront():void|
|||+--> | moveback():void |
|||| +-----+
||||
||||
||||
|||| +-----+
|||| | | | | |
|||| +-----+ +-----+ +-----+ +-----+
|||| |moveleft| |moveright| |movefront| |moveback|
|||| +-----+<>. .<>+-----+ +-----+<>. .<>+-----+
|||| | execute| | | execute | | execute | | | execute|
|||| +-----+ | | +-----+ +-----+ | | +-----+
|||+-----+ | | |
||+-----+ | |
|+-----+ |
+-----+

```

Code Example

```

package command;
sub new {
my ($class,$robot)=@_;
bless {robot=>$robot},$class;
}
1;
<<command.pm>>

```

```

package moveleft;
use base qw(command);
sub execute {
my $ref=shift;
$ref->{robot}->moveleft();
}
1;
<<moveleft.pm>>

```

```

package moveright;
use base qw(command);
sub execute {
my $ref=shift;
$ref->{robot}->moveright();
}
1;
<<moveright.pm>>

```

```

package movefront;
use base qw(command);
sub execute {
my $ref=shift;
$ref->{robot}->movefront();
}
1;
<<movefront.pm>>

```



```
package moveback;
use base qw(command);
sub execute {
my $ref=shift;
$ref->{robot}->moveback();
}
1;
<<moveback.pm>>
```

```
package robot;
sub new {
my $class=shift;
bless {},$class;
}
sub moveleft {
print "moved left\n";
}
sub moveright {
print "moved right\n";
}
sub movefront {
print "moved front\n";
}
sub moveback {
print "moved back\n";
}
1;
<<robot.pm>>
```

```
package invoker;
sub new {
my $class=shift;
bless {order=>undef},$class;
}
sub takeorder {
my ($ref,@actions)=@_;
push @{$ref->{order}},@actions;
}
sub perform {
my $ref=shift;
map($->execute,@{$ref->{order}});
}
1;
<<invoker.pm>>
```

```
use strict;
use warnings;
```

```
use invoker;
use moveleft;
use moveright;
use movefront;
use moveback;
use robot;
```

```
my $robot=new robot;
my $invoker=new invoker;
$invoker->takeorder(new moveleft($robot),new moveleft($robot),new movefront($robot),new
```

Iterator

In a collection class when each element of the collection (i.e. Array, list, tree, etc.) needs to be accessed sequentially and in a collection type independent manner an iterator pattern is used. Interfaces provided through the pattern are the same for all collection classes, can be an array list tree or any other, and it is independent of the class internal representation.

For example a list user should be able to iterate the list in the same manner as an array user. In CD player user does not need to worry about in what format songs are stored he just presses the next button in order to go to the next song.



Code Example

```
package aggregate;
sub new {
my ($class,$objectref)=@_;
bless $objectref,$class;
}
1;
<<aggregate.pm>>

package iterator;
sub new {
my ($class,$aggregate)=@_;
bless {AGGREGATE=>$aggregate, INDEX=>0, BEGIN=>0, END=>-1}, $class;
}
1;
<<iterator.pm>>

package stack;
use base qw(aggregate);
use stackiterator;
sub new {
my $class=shift;
$class->SUPER::new([]);
}
sub createiterator {
my $ref=shift;
stackiterator->new($ref);
}
1;
<<stack.pm>>

package stackiterator;
use base qw(iterator);
sub getfirst {
my $ref=shift;
${$ref->{AGGREGATE}}[$ref->{INDEX}=$ref->{END}];
}
sub getnext {
my $ref=shift;
${$ref->{AGGREGATE}}[--$ref->{INDEX}];
}
sub end {
my $ref=shift;
($ref->{INDEX} < $ref->{BEGIN})?1:0;
}
sub additem {
my ($ref,$item)=@_;
print "adding to stack:",$item,"\n";
push @{$ref->{AGGREGATE}}, $item;
$ref->{END}+=1;
}
sub removeitem {
my $ref->shift;
print "removing from stack:",${$ref->{AGGREGATE}}[$ref->{END}],"\n";
--$ref->{END};
}
1;
```

```
<<stack.pm>>

package queue;
use base qw(aggregate);
use queueiterator;
sub new {
my $class=shift;
$class->SUPER::new([]);
}
sub createiterator {
my $ref=shift;
queueiterator->new($ref);
}
1;
<<queue.pm>>

package queueiterator;
use base qw(iterator);
sub new {
my $class=shift;
my $ref=$class->SUPER::new;
($ref->{SEGMENTSIZ},$ref->{CURRENTSEGMENTBEGIN},$ref->{CURRENTSEGMENTEND},$ref->{SEGMENTINDEX},$ref->{SEGMENTCOUNT},$ref->{SEGMENT})=(5,0,-1,-1,0,[0]);
$ref;
}
sub getfirst {
my $ref=shift;
$ref->{SEGMENTINDEX}=$ref->{CURRENTSEGMENTBEGIN};
${$ref->{AGGREGATE}}[$ref->{INDEX}=$ref->{BEGIN}];
}
sub getnext {
my $ref=shift;
if(!(($ref->{INDEX}+1)%($ref->{SEGMENTSIZ}))) {
++$ref->{SEGMENTINDEX};
$ref->{INDEX}=${$ref->{SEGMENT}}[$ref->{SEGMENTINDEX}]*$ref->{SEGMENTSIZ};
--$ref->{INDEX};
}
${$ref->{AGGREGATE}}[++$ref->{INDEX}];
}
sub additem {
my ($ref,$item)=@_;
print "adding to queue:",$item,"\n";
if(!(($ref->{END}+1)%($ref->{SEGMENTSIZ}))) {
++$ref->{CURRENTSEGMENTEND};
${$ref->{SEGMENT}}[$ref->{CURRENTSEGMENTEND}]=$ref->{CURRENTSEGMENTEND} if !defined ${$ref->{SEGMENT}}[$ref->{CURRENTSEGMENTEND}];
$ref->{END}=${$ref->{SEGMENT}}[$ref->{CURRENTSEGMENTEND}]*$ref->{SEGMENTSIZ}-1;
}
${$ref->{AGGREGATE}}[++$ref->{END}]=$item;
}
sub removeitem {
my $ref=shift;
if(!(($ref->{BEGIN}+1)%($ref->{SEGMENTSIZ}))) {
++$ref->{CURRENTSEGMENTBEGIN};
$ref->{BEGIN}=${$ref->{SEGMENT}}[$ref->{CURRENTSEGMENTBEGIN}]*$ref->{SEGMENTSIZ};
my $i=0;
while($i!=$ref->{SEGMENTCOUNT}) {
${$ref->{SEGMENT}}[$i]+=${$ref->{SEGMENT}}[$i+1];
${$ref->{SEGMENT}}[$i+1]=${$ref->{SEGMENT}}[$i]-${$ref->{SEGMENT}}[$i+1];
```

```

${$ref->{SEGMENT}}[$i]==${$ref->{SEGMENT}}[$i+1];
++$i;
}
$ref->{CURRENTSEGMENTEND}--1;
$ref->{CURRENTSEGMENTBEGIN}--1;
}else {
++$ref->{BEGIN};
}
}
sub end {
my $ref=shift;
($ref->{END} < $ref->{INDEX})?1:0;
}
1;
<<queueiterator.pm>>

```

```

use strict;
use warnings;
use stack;
use queue;
my $stackiteratorref=new stack->createiterator;
my $queueiteratorref=new queue->createiterator;
for my $iterator ($stackiteratorref,$queueiteratorref) {
$iterator->additem(1);
$iterator->additem(2);
$iterator->additem('a');
$iterator->additem('b');
$iterator->additem('c');
$iterator->additem('d');
$iterator->additem('e');
for(my $item=$iterator->getfirst; !$iterator->end; $item=$iterator->getnext) {
print "$item ";
}
print "\n";
}
<<main.pl>>

```

---data---

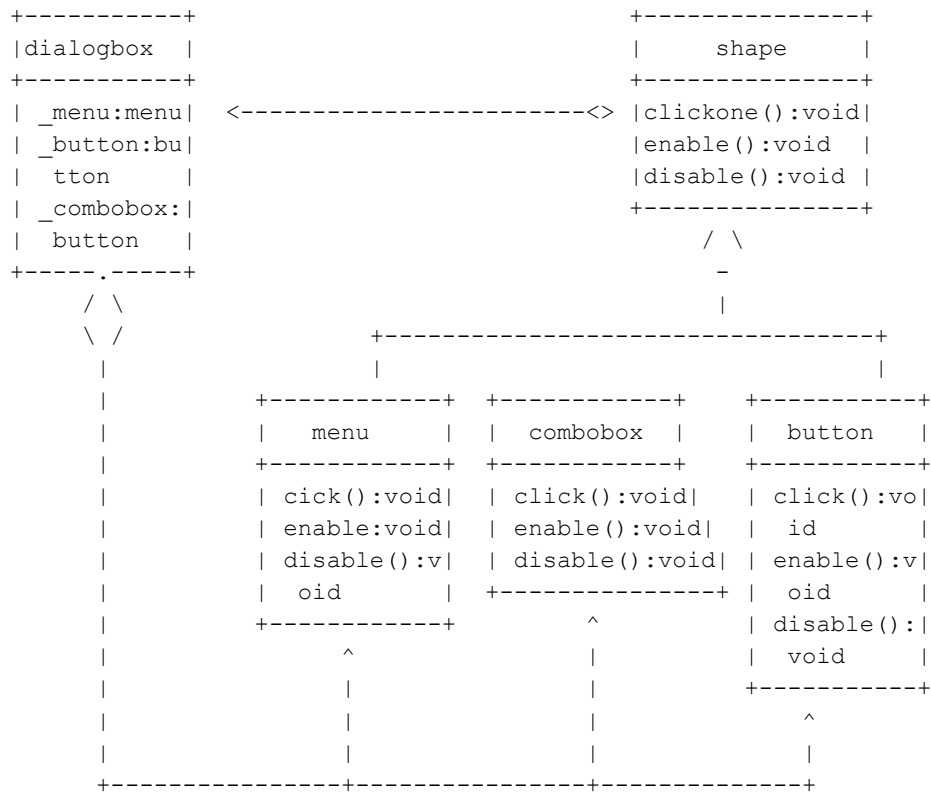
```

adding to stack:1
adding to stack:2
adding to stack:a
adding to stack:b
adding to stack:c
adding to stack:d
adding to stack:e
e d c b a 2 1
adding to queue:1
adding to queue:2
adding to queue:a
adding to queue:b
adding to queue:c
adding to queue:d
adding to queue:e
1 2 a b c d e
-----

```

Mediator

When a group of entities exists and change in one state effect other makes entities are tightly coupled this way. Adding or removing an entity will lead to change in the code of all the entities and when the group is big it is difficult to handle this situation. The idea is to introduce a mediator and all separate entities in the group would report to the mediator. This makes each entity in the group independent of each other and it is the mediator who decides for updating among entities. Telephone exchange (switch) connects N users, whereas connecting each other without switch would be practically impossible. Head of a team is actually a mediator.



Code Example

```

package dialogbox;
sub new {
my $class=shift;
bless {},$class;
}
sub registermenu {
my ($ref,$shape)=@_;
$ref->{MENU}=$shape;
}
sub registerbutton {
my ($ref,$shape)=@_;
$ref->{BUTTON}=$shape;
}
sub registercombobox {
my ($ref,$shape)=@_;
$ref->{COMBOBOX}=$shape;
}
sub informmenu {
my ($ref,$shape)=@_;
print "dialogbox::menu clicked id:",$shape->id,"\n";
}
sub informbutton {

```

```
my ($ref,$shape)=@_;
print "dialogbox::button clicked id:",$shape->id,"\n";
}
sub informcombobox {
my ($ref,$shape)=@_;
print "dialogbox::combobox clicked id:",$shape->id,"\n";
}
1;
<<dialogbox.pm>>
```

```
package shape;
sub new {
my ($class,$id,$dialogbox)=@_;
bless {ID=>$id,MEDIATOR=>$dialogbox},$class;
}
sub id {
my $ref=shift;
$ref->{ID};
}
1;
<<shape.pm>>
```

```
package menu;
use base qw(shape);
sub new {
my ($class,$id,$dialogbox)=@_;
my $ref=$class->SUPER::new($id,$dialogbox);
$ref->{MEDIATOR}->registermenu($ref);
$ref;
}
sub click {
my $ref=shift;
$ref->{MEDIATOR}->informmenu($ref);
}
1;
<<menu.pm>>
```

```
package button;
use base qw(shape);
sub new {
my ($class,$id,$dialogbox)=@_;
my $ref=$class->SUPER::new($id,$dialogbox);
$ref->{MEDIATOR}->registerbutton($ref);
$ref;
}
sub click {
my $ref=shift;
$ref->{MEDIATOR}->informbutton($ref);
}
1;
<<button.pm>>
```

```
package combobox;
use base qw(shape);
sub new {
my ($class,$id,$dialogbox)=@_;
my $ref=$class->SUPER::new($id,$dialogbox);
$ref->{MEDIATOR}->registercombobox($id,$dialogbox);
$ref;
}
```

```

}
sub click {
my $ref=shift;
$ref->{MEDIATOR}->informcombobox($ref);
}
1;
<<combobox.pm>>

use strict;
use warnings;
use dialogbox;
use menu;
use button;
use combobox;
my $dialogboxref=new dialogbox;
new menu('menu1',$dialogboxref)->click;
new button('button1',$dialogboxref)->click;
new combobox('combobox1',$dialogboxref)->click;
<<main.pl>>
---data---
dialogbox::menu clicked id:menu1
dialogbox::button clicked id:button1
dialogbox::combobox clicked id:combobox1
-----

```

Arithmetic expression calculation:

```

+-----+
| mexpression |
+-----+
v          <> +-----+
|          | evaluate(exp |
|          | resson:stri |
|          | ng          |
+-----+
/ \
+-----+
| mediator |
+-----+
| _mul:multiplier |
| _div:division   |
| _add:addition   |
| _sub:subtraction|
+-----+
| evaluate(express|
| ion):void       |
+-----+
/ \
\ / +-----+
| | | | | |
| +-----+ +-----+ +-----+ +-----+
| |multiplier| |divider| |subtractor| | adder |
| +-----+ +-----+ +-----+ +-----+
| | evaluate(e| | evaluate(| | evaluate(e| | evaluate(e|
| | xpression)| | expression| | xpression)| | xpression|
| | :string | | ):string| | :string | | :string |
| +-----+ +-----+ +-----+ +-----+
| ^ ^ ^ ^
| | | |
+-----+

```


Code Example

```
package mediator;
sub new {
  my ($class,$multiplier,$divider,$subtractor,$adder)=@_;
  my $ref=bless {MULTIPLIER=>$multiplier,DIVIDER=>$divider,SUBTRACTER=>$subtractor,ADDER=>$adder},$class;
  map{$_->mediator($ref)} ($multiplier,$divider,$subtractor,$adder);
  $ref;
}
sub informmultiplier {
  my ($ref,$expression)=@_;
  $ref->{SUBTRACTER}->evaluate($expression);
}
sub informdivider {
  my ($ref,$expression)=@_;
  $ref->{MULTIPLIER}->evaluate($expression);
}
sub informsubtractor {
  my ($ref,$expression)=@_;
  $ref->{ADDER}->evaluate($expression);
}
sub informadder {
  my ($ref,$expression)=@_;
  $expression;
}
sub evaluate {
  my ($ref,$expression)=@_;
  print "expression:",$expression,"\n";
  $ref->{DIVIDER}->evaluate($expression);
}
1;
<<mediator.pm>>

package expression;
sub new {
  my ($class,$operator)=@_;
  bless {operator=>$operator},$class;
}
sub mediator {
  my ($ref,$mediator)=@_;
  $ref->{MEDIATOR}=$mediator;
}
sub evaluate {
  my ($ref,$expression)=@_;
  while($expression=~/(-?\d+)(\ref->{operator})(-?\d+)/) {
    $expression=join('',$`,eval "$1$2$3",$');
  }
  $expression;
}
1;
<<expression.pm>>

package multiplier;
use base qw(expression);
sub new {
  my $class=shift;
  $class->SUPER::new('\*');
```

```

}
sub evaluate {
my ($ref,$expression)=@_;
print "multiplier,expression:",$expression,"\n";
$ref->{MEDIATOR}->informmultiplier($ref->SUPER::evaluate($expression));
}
1;
<<multiplier.pm>>

package divider;
use base qw(expression);
sub new {
my $class=shift;
$class->SUPER::new('/');
}
sub evaluate {
my ($ref,$expression)=@_;
print "divider,expression:",$expression,"\n";
$ref->{MEDIATOR}->informdivider($ref->SUPER::evaluate($expression));
}
1;
<<divider.pm>>

package subtracter;
use base qw(expression);
sub new {
my $class=shift;
$class->SUPER::new('-');
}
sub evaluate {
my ($ref,$expression)=@_;
print "subtracter,expression:",$expression,"\n";
$ref->{MEDIATOR}->informsubtracter($ref->SUPER::evaluate($expression));
}
1;
<<subtracter.pm>>

package adder;
use base qw(expression);
sub new {
my $class=shift;
$class->SUPER::new('+');
}
sub evaluate {
my ($ref,$expression)=@_;
print "adder,expression:",$expression,"\n";
$ref->{MEDIATOR}->informadder($ref->SUPER::evaluate($expression));
}
1;
<<adder.pm>>

use strict;
use warnings;
use multiplier;
use divider;
use subtracter;
use adder;
use mediator;
print "result:", new mediator(new multiplier,new divider,new subtracter,new adder)->evalua

```

There are scenario when a class (originator) changes its state and there has to be an option to bring the class to certain state that a class had been to in the past. It should support kind of undo operation. The class can itself contain all the states it happened to be in the past or a separate opaque class (memento) can be introduced to keep the state of the originating class. A caretaker who handles the originator's behavior keeps various states of the originator in opaque memento and when undo operation is required it sets the originator's state to state stored in a particular memento object. For example a simple adder adds two numbers and gives the result. There can be a need that the user needs to see previous calculations and in this case caretaker would let originator create many mementos of past calculations and when undo is required previous mementos is set as new state of the originator. Ministers are mementos for a king's action and decisions and when Required they make king remember the past actions.

caretaker		calculator
save(firstnumber: int, secondnumber: int):void	<>--+ +----	backup(firstno:int, secondno:int):void
firstnumber():int		restore(state:memnto)
secondnumber():int		getresult():int
		setfirstnumber(no:int):void
		setsecondnumber(no:int):void
		/ \
		-
memento	<--+	
store(firstno:int, secondno:int):void	<-----+	intadder
firstno():int		firstnumber:int
secondno():int		secondnumber:int

Code Example

```
package adder;
sub new {
my ($class,$firstnumber,$secondnumber)=@_;
bless {FIRSTNUMBER=>$firstnumber,SECONDNUMBER=>$secondnumber},$class;
}
sub add {
my $ref=shift;
print "originator(adder)::add firstnumber,secondnumber:",$_[0],":",$_[1],"\n" if scalar @_;
print "originaator(adder)::add firstnumber,secondnumber:",$ref->{FIRSTNUMBER},":", $ref->{SECONDNUMBER},"\n" if !scalar @_;
(scalar @_)?($ref->{FIRSTNUMBER}=shift)+($ref->{SECONDNUMBER}=shift):$ref->{FIRSTNUMBER}+$ref->{SECONDNUMBER};
}
sub backup {
my ($ref,$caretaker)=@_;
print "originator(adder)::backup firstnumber,secondnumber:",$ref->{FIRSTNUMBER},":", $ref->{SECONDNUMBER},"\n";
$caretaker->save($ref->{FIRSTNUMBER},$ref->{SECONDNUMBER});
}
sub restore {
my ($ref,$caretaker)=@_;
print "originator(adder)::restore firstnumber,secondnumber:",$caretaker->firstnumber,":", $caretaker->secondnumber,"\n";
($ref->{FIRSTNUMBER},$ref->{SECONDNUMBER})=($caretaker->firstnumber,$caretaker->secondnumber);
}
1;
<<adder.pm>>

package caretaker;
my $caretakerref=undef;
sub caretaker {
$caretakerref=__PACKAGE__->new if !defined $caretakerref;
$caretakerref;
}
sub new {
my $class=shift;
bless {MEMENTOSTACK=>undef, TOP=>-1, INDEX=>-1, MAXCOUNT=>5},$class;
}
sub getnext {
my $ref=shift;
${$ref->{MEMENTOSTACK}}[($ref->{TOP}+1)%($ref->{MAXCOUNT})];
}
sub get {
my $ref=shift;
${$ref->{MEMENTOSTACK}}[$ref->{INDEX}=$ref->{TOP}];
}
sub push {
my ($ref,$value)=@_;
print "caretaker,adding memento to queue",$value->{FIRSTNUMBER},":", $value->{SECONDNUMBER},"\n";
${$ref->{MEMENTOSTACK}}[$ref->{INDEX}=$ref->{TOP}=( $ref->{TOP}+1)%($ref->{MAXCOUNT})]=$value;
}
sub getprevious {
my $ref=shift;
${$ref->{MEMENTOSTACK}}[($ref->{INDEX}==0)?$ref->{INDEX}=$ref->{MAXCOUNT}-1:--$ref->{INDEX}];
}
```

```

1;
<<caretaker.pm>>

package memento;
use base qw(caretaker);
sub new {
my ($class,$firstnumber,$secondnumber)=@_;
bless {FIRSTNUMBER=>$firstnumber,SECONDNUMBER=>$secondnumber},$class;
}
sub firstnumber {
my $ref=shift;
(scalar @_)?$ref->{FIRSTNUMBER}=shift:$ref->{FIRSTNUMBER};
}
sub secondnumber {
my $ref=shift;
(scalar @_)?$ref->{FIRSTNUMBER}=shift:$ref->{SECONDNUMBER};
}
sub save {
my ($ref,$firstnumber,$secondnumber)=@_;
my $mementoref=$ref->caretaker->getnext;
if(!defined $ref->{FIRSTNUMBER}) {
($ref->{FIRSTNUMBER},$ref->{SECONDNUMBER})=($firstnumber,$secondnumber);
if(!defined $mementoref) {
$ref->caretaker->push($ref);
}else {
$ref->caretaker->push($mementoref);
}}else {
if(!defined $mementoref) {
$ref->caretaker->push(__PACKAGE__->new($firstnumber,$secondnumber));
}else {
$mementoref->firstnumber($firstnumber);
$mementoref->secondnumber($secondnumber);
}
}
}
sub previous {
my $ref=shift;
$ref->caretaker->getprevious;
}
sub get {
my $ref=shift;
$ref->caretaker->SUPER::get;
}
1;
<<memento.pm>>

use strict;
use warnings;

use memento;
use adder;
my $adderref=new adder;
my $caretaker=new memento;
print "result:",$adderref->add(10,20),"\\n";
print "result:",$adderref->add(20,30),"\\n";
$adderref->backup($caretaker);
print "result:",$adderref->add(1000,20),"\\n";
$adderref->backup($caretaker);
print "result:",$adderref->add(420,-20),"\\n";

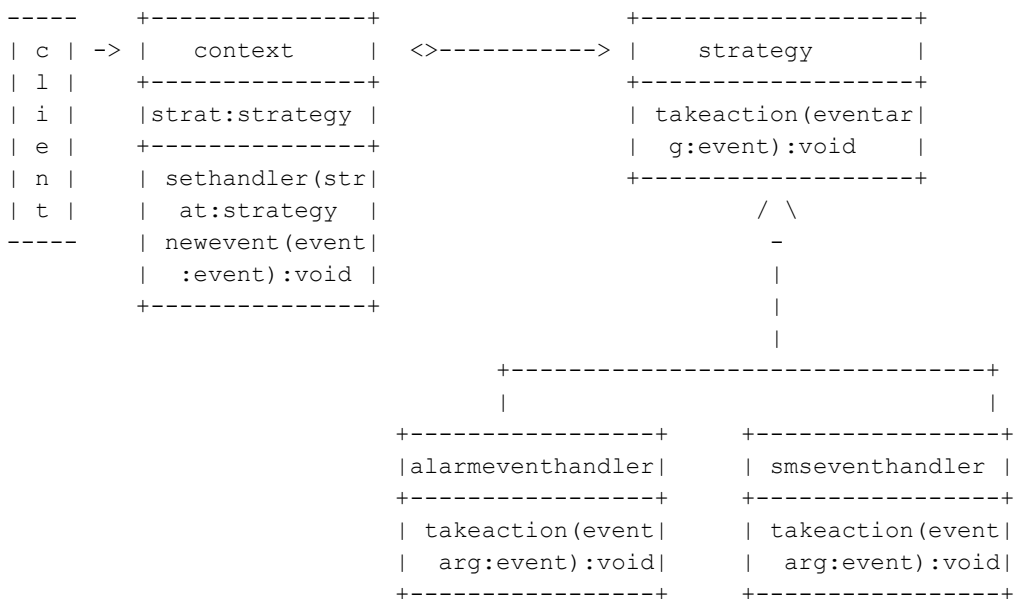
```

```
$adderref->backup($caretaker);
$adderref->add(10,-100000000);
$adderref->restore($caretaker->get);
print "result:",$adderref->add;
<<main.pl>>
```

```
---data---
originator(adder)::add firstnumber,secondnumber:10:20
result:30
originator(adder)::add firstnumber,secondnumber:20:30
result:50
originator(adder)::backup firstnumber,secondnumber:20:30
caretaker,adding memento to queue20:30
originator(adder)::add firstnumber,secondnumber:1000:20
result:1020
originator(adder)::backup firstnumber,secondnumber:1000:20
caretaker,adding memento to queue1000:20
originator(adder)::add firstnumber,secondnumber:420:-20
result:400
originator(adder)::backup firstnumber,secondnumber:420:-20
caretaker,adding memento to queue420:-20
originator(adder)::add firstnumber,secondnumber:10:-100000000
originator(adder)::restore firstnumber,secondnumber:420:-20
originator(adder)::add firstnumber,secondnumber:420:-20
result:400
-----
```

Strategy

A behavior of a class may contain static behavior and changeable behavior. Instead of hardcoding the changeable behavior, it is kept abstract data type followed by getting the implementation of abstract at runtime. This kind of abstract data type is addressed through strategy pattern. Abstract data type, then subclassed into various implementations, which is passed to the main class at run time, making the possibility to change the behavior at run time. I.e. an event manager when receives an event it passes it through an algorithm which parses the event structure and takes necessary action, i.e. logging it, raising alarm, etc. This algorithm now can be placed in strategy abstract type so that it may vary and event manager code remain unchanged leading to new and better algorithm run in this fashion. A person plans (makes strategy) for his kid's life that at a particular age, he has to go to school and then he has to take a job then marriage... But he does not know the names.



Code Example

```
package eventmanager;
sub new {
my ($class,$handler)=shift;
bless {STRAT=>$handler},$class;
}
sub sethandler {
my ($ref,$handler)=@_;
my $oldhandler=$ref->{sTRAT};
$ref->{STRAT}=$handler;
$oldhandler;
}
sub handleevent {
my ($ref,$event)=@_;
$ref->{STRAT}->takeaction($event);
}
1;
<<eventmanager.pm>>

package eventhandler;
sub new {
my $class=shift;
bless {},$class;
}
1;
<<eventhandler.pm>>

package loggerhandler;
use base qw(eventhandler);
sub takeaction {
my ($ref,$event)=@_;
print "loggerhandler, logging to a file, event:",$event,"\n";
}
1;
<<loggerhandler.pm>>

package smshandler;
use base qw(eventhandler);
sub takeaction {
my ($ref,$event)=@_;
print "smshandler, sending sms, event:",$event,"\n";
}
1;
<<smshandler.pm>>

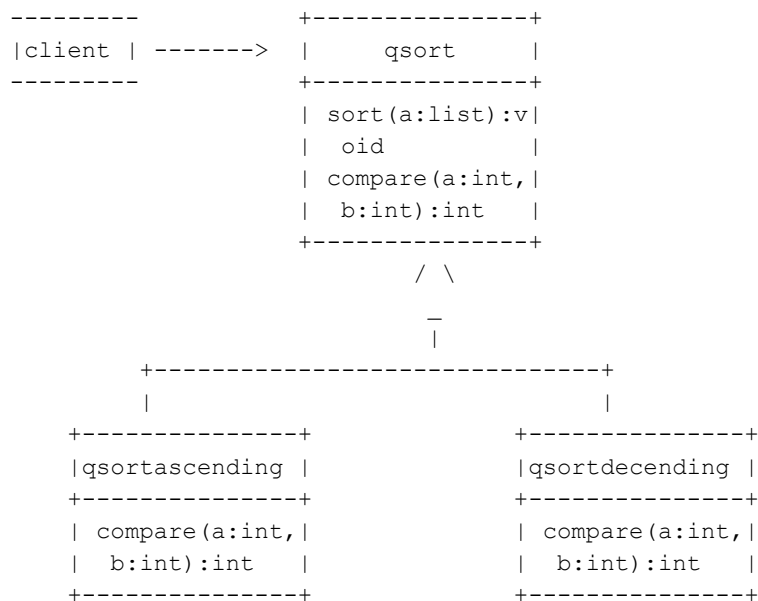
use strict;
use warnings;

use loggerhandler;
use smshandler;
use eventmanager;
my $eventmanager=new eventmanager;
$eventmanager->sethandler(new loggerhandler);
$eventmanager->handleevent("server down");
$eventmanager->sethandler(new smshandler);
$eventmanager->handleevent("server down");
<<main.pl>>
```

```
---data---
loggerhandler, logging to a file, event:server down
smshandler, sending sms, event:server down
-----
```

Template Method

As per the definition of template, a template draws an architecture of something and let implement or implement the architecture from his perspective. A template is kind of a starting point. Similarly, template method draws the layout of behaviors. The class containing the template method later subclasses in order to fill the architecture to some real examples. For example, someone writes an algorithm to sort an array or list. Sorting can happen in at least two ways ascending and descending. Rather than mentioning these in the algorithm he just writes the architecture and leave a hook point where ascending and descending specific codes in specific subclasses would make it two separate algorithms ascending sorting and descending sorting. A person can work on one task or two or three, but if he needs to work on 100s of tasks, then either he cannot, or he has to find a common thing among them so that they would differ in very few points.



Code Example

```
package qsort;
sub new {
my $class=shift;
bless {},$class;
}
sub sort {
my ($ref,@a)=@_;
print "array to be sorted:",map("$_",",",@a),"\\n";
$ref->sorti(0,$#a,\\@a);
print "sorted array:",map("$_",",",@a),"\\n";
}
sub sorti {
my ($ref,$m,$n,$a)=@_;
if($m<$n) {
$i=$m;$j=$n+1;
do {
do {
$i=$i+1;
}while $i<=$n && $ref->compare1(${$a}[$i],${$a}[$m]);
do {
```



```

$j=$j-1;
}while $ref->compare2($$a[$j],$$a[$m]);
if($i<$j) {
$t=$$a[$i];$$a[$i]=$$a[$j];$$a[$j]=$t;
}
}while $i<$j;
$t=$$a[$j];$$a[$j]=$$a[$m];$$a[$m]=$t;
$ref->sorti($m,$j-1,$a);
$ref->sorti($j+1,$n,$a);
}
}
1;
<<qsort.pm>>

```

```

package qsortascending;
use base qw(qsort);
sub compare1 {
my ($ref,$a,$b)=@_;
($a<$b)?1:0;
}
sub compare2 {
my ($ref,$a,$b)=@_;
($a>$b)?1:0;
}
1;
<<qsortascending.pm>>

```

```

package qsortdescending;
use base qw(qsort);
sub compare1 {
my ($ref,$a,$b)=@_;
($a>$b)?1:0;
}
sub compare2 {
my ($ref,$a,$b)=@_;
($a<$b)?1:0;
}
1;
<<qsortdescending.pm>>

```

```

use strict;
use warnings;

use qsortascending;
use qsortdescending;
new qsortascending->sort(1,5,3,7,2,10,20,15,17,11);
new qsortdescending->sort(1,5,3,7,2,10,20,15,17,11);
<<main.pl>>

```

---data---

```

array to be sorted:1,5,3,7,2,10,20,15,17,11,
sorted array:1,2,3,5,7,10,11,15,17,20,
array to be sorted:1,5,3,7,2,10,20,15,17,11,
sorted array:20,17,15,11,10,7,5,3,2,1,

```

Summary

In this article various aspects of design patterns are discussed with addition of placing the examples in Perl. Article is written in a way that it would be equally useful to programmers in other languages. In addition to elaborate examples emphasis is also given in general design concepts. This article discusses GOF 23 design patterns based on creational, structural and behavioural subpatterns. All three types of patterns are provided through more in-depth view as how they are distributed, i.e. creational pattern is distributed among new object creation and cloning existing object whereas structural patterns are distributed among inheritance, composition and inheritance+composition and behavioural patterns are distributed among recursive and non-recursive way. These design patterns also cover the generic programming styles.

Technical Interviewing Technique: Looking for an Intuitive Narrative

by Soumen Sarkar and Jeff Edmonds

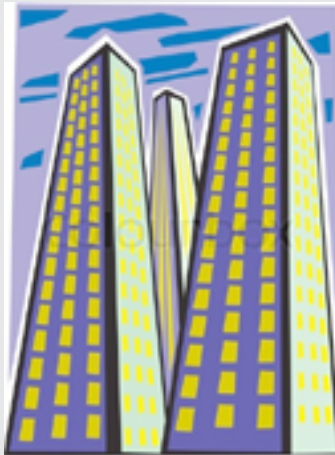
I wanted to write this article to contemplate upon a seemingly vexing problem: In a technical interview, which is a limited contract opportunity, how does the interviewer measure candidate's power of reason and skill of communication on a rationalistic and intuitive basis?

We came to the conclusion that the whole thing depends upon:

- Candidate's preparedness (technical and communication)
- Interviewer's preparedness (interviewing and communication)
- Luck (both on the part of candidate and interviewer)

We think we have a good technical problem, which we hope to be able to do a convincing articulation of the above three.

The Problem



You're standing in front of a 100-story building with two identical bowling balls. You've been tasked with testing the bowling balls' resilience. The building has a stairwell with a window at each story from which you can (conveniently) drop bowling balls. To test the bowling balls you need to find the first floor at which they break. It might be the 100th floor or it might be the 37th floor, but if it breaks somewhere in the middle you know it will break at every floor above. If a ball is not broken, please reuse the ball. Devise an algorithm, which guarantees you'll find the first floor at which one of your bowling balls will break. You're graded on your algorithm's worst-case running time.

Interview Structure

This article will propose a guided structure for technical interview in the backdrop of this particular question. The guidance from the interviewer, in my humble opinion, is quite crucial so that the interviewee's power of reason and communication skill can be drawn out and assessed while the interviewer retains control of the engagement experience. In other words, *the interviewer cannot sit back and relax!* We are proposing the following structure:

- Interviewer identifies the constraint of two bowling balls (space complexity constraint) and states that s/he will relax this constraint initially with gradual tightening
- Interviewer identifies the constraint of the number of tries (time complexity constraint) and states that s/he is looking for minimizing worst cases
- The interviewer states that the interview is interactive and advises the candidate to listen carefully and ask questions
- The interviewer then states that following four cases will be considered in an interactive manner:

Case #	Space Complexity	Worst Case Time Complexity
Case 1	Can use only one ball	How many tries?
Case 2	Can use as many balls	How many tries?
Case 3	Can use only two balls	How many tries?
Case 4	Can use only three balls	How many tries?

One Ball Case

One ball case is quite important due to following reasons:

- cebreaker (interviewee starts talking about a trivial case)

This is a trivial case. Since the algorithm must work for all floors, the only option is to try floor 1 to 100 one-by-one. This also removes any irrelevant details from a candidate's mind like to repeat dropping may make the ball brittle.

Let's assign 10 points out of 100 for this case.

As Many Balls Case

Now that, candidate is talking, let's try to engage on as many balls case. Does the candidate say 100 balls, or 50 balls or something less? If the candidate does not zero on binary search algorithm within five minutes, it is not looking good! Binary search is quite obvious – from a divide and conquer point of view. First drop one ball from floor 50 ($= 100/2$). If it breaks, then use other balls for lower half (floors 1 through 49) using divide and conquer to halve the problem size at each drop. If the ball does not break then try upper half (floors 51 to 100) using divide and conquer to halve the problem size at each drop. If the candidate gets binary search then you may ask why binary search is applicable to this problem? What is sorted (since binary search is applicable on sorted data structure)?

So the answer is, we need maximum $\text{ceil}(\log_2(100)) = \text{ceil}(6.x) = 7$ balls/tries. Also note that there is no separation of time complexity (number of tries) and space complexity (number of balls) in this case.

Time Complexity = Space Complexity = 7

Let's assign 40 points out of 100 for this case. So if the candidate has made till this point, s/he is half way through ($10 + 40 = 50$ points out of 100).

Two Balls Case

It is time to separate space and time complexity. We constrain the space complexity to two (balls). It should be obvious that time complexity is more than two – come on there are 100 floors! Actually, it has to be at least seven (see previous section). Ask the candidate on the minimum bound of time complexity in this case and look for seven. If the candidate chooses a step size of 14 then there needs to be at least 7 tries ($100/14 = 7.X$) till the first ball breaks or it is certain that it will break. Let's take an example; let's say the ball breaks

at 100 (the topmost floor). In this case, there will be 7 tries (14, 28, 42,..., 84, 98) and then the candidate can try 99 and 100 \Rightarrow total $7 + 2 = 9$ tries. If the ball breaks in 97, then the number of tries = 7 (the first ball breaks in floor 98) + 13 (try 2nd ball 85, 86,..., 96, 97) = 20. At this point the candidate could recognize that there are two variables at play:

- Interval size
- Number of intervals

The following equation may help:

$$(\text{Interval size}) \times (\text{Number of intervals}) \geq 100$$

For example, if the interval size is 4, then we will need 25 intervals (not good). On the other hand, the interval size of 25 is not desirable as well since if the first ball breaks, then we do need to try linearly within the last interval. We have a situation where the product of two variables is fixed ($= 100$) and we need to minimize the sum of the two variables. This is achieved by setting the two variables equal to each other:

$$\text{Interval size} = \text{Number of intervals} = N$$

Then we have,

$$N \times N = 100$$

$$\rightarrow N = 10 \text{ (square root of 100)}$$

A quick calculation shows that maximum number of tries is $10 + 9 = 19$. Checking back, if the answer is 99 then we lose the first ball at floor 100 (10, 20,..., 100) and then we need to try linearly (91, 92,..., 99) with the remaining balls.

Let's assign 20 points out of 100 for this sub-case. If the candidate has made out this far, then the score is at 70 ($10 + 40 + 20$).

Two Balls Case (Optimized)

At this point, the candidate could be told that s/he did quite well. However, number of tries could be improved through a better strategy. Assuming the optimum number is N , how can we derive it? At this point, the narrative could change. We do not know how the candidate will respond. May be this problem is already known to the candidate? However, this complication does not change the intent of this article. The intent of this article is to show how a multi stage problem (with increasing level of sophistication) could possibly used to assess interviewee's critical thinking ability.

Getting back to the problem... this is what we know about calculating N . Let's start with our initial assertion that we have a strategy of N ball drops. Let's forget for a moment that we have 100 floors. Let's say we have N floors – what is the best strategy with the two balls? We think this should be obvious – drop the first ball from floor N . Let's say it breaks – in that case, next ball can be used linearly (one-by-one) for floors 1 to $(N - 1)$. So if the ball breaks at floor N or lower, we are still maintaining the logical consistency that we have a strategy of N ball drops.

$$\text{Invariant} \rightarrow 1 + (N - 1) = N$$

What happens if the ball does not break? Well, in that case we can do the second drop from the floor $(N + (N - 1))$. Why are we going $(N - 1)$ floor above floor N ? This is because at this point we have used two drops and if the ball breaks at floor $(N + (N - 1))$, we still can use remaining $(N - 2)$ tries to check intermediate floors.

$$\text{Invariant} \rightarrow 1 + 1 + (N - 2) = N$$

Again, we are still maintaining the logical consistency that we have a strategy of N ball drops. This act maintaining of the initial assertion that we have a strategy of N drops and not violating this assumption is called *Maintaining The Invariant* (the heart of any algorithm). A crucial thing to observe now is.

We are going higher (from N th floor to $N + N - 1$ floor) while maintaining the invariant that maximum number of tries is $\leq N$:

Loop Invariant
Proving Your Algorithms

- Start Small (initial condition)
- Make Progress ($N \rightarrow N+1$ or $N \rightarrow N-1$)
- Maintain Invariant

How long this can continue? Well, this can continue till the sequence comes down to last term 1:

N ($N - 1$) on top of N ($N - 2$) on top of ($N + (N - 1)$) (... 3 on top of previous (2 on top of previous (1 on top of previous.

If we add all these intervals, at the end, we must reach the 100th floor. So we have the following equation:

$$N + (N - 1) + (N - 2) + \dots + 3 + 2 + 1 \geq 100$$

$$\rightarrow N(N + 1) / 2 \geq 100$$

$$\rightarrow N = 13.X$$

$$\rightarrow N = 14$$

Let's assign 30 points out of 100 for this sub-case. Checking back for floor 100 case, we go up to floor 99 in 11 tries ($14 + 13 + 12 + 11 + 10 + 9 + 8 + 7 + 6 + 5 + 4$) and the ball break at floor 100 (total 12 ($11 + 1$) tries.

If the candidate has made out this far, then the score is at 100 ($10 + 40 + 20 + 30$). Full score!!

Three Balls Case

Whew! This has been quite a journey. However, we do not need to stop here. Let's consider 20-point bonus question:

What If You Can Drop Three Balls?

My response would be to use one ball to halve the problem size. Drop one ball from 50th floor and expect it to break. If it breaks, then use other two balls for floors 1 through 49 with two balls optimized algorithms described in the previous section. In that case:

$$N(N + 1) / 2 = 49$$

$$\rightarrow N = 9.X$$

$$\rightarrow N = 10$$

Counting the first ball drop (to halve the problem space), we have to add one:

$$N = 10 + 1 = 11$$

If the ball does not break, continue using the third ball to halve the problem size and when the third ball breaks, use other two balls as per two-ball-optimized algorithm. At this point, it is quite evident what to do with four or five balls. For example, for four balls, the answer is $N = 7 + 2 = 9$. If the candidate gets this as well then s/he gets 120 (full score+ 20 bonus points).

This situation is screaming for the candidate to be hired if the narrative appeals to both logic and feeling. The art of knowing what the candidate knows takes us to the next section.

Objective Epistemology by Ayn Rand

There is a branch of philosophy on pondering the business of knowing: Objective Epistemology. Ayn Rand published Introduction to Objectivist Epistemology, a monograph on the Objectivist theory of concepts in 1967.

Objective Epistemology by Ayn Rand

How do we know what we know? Is reason a reliable source of knowledge – or is it superseded by mystical revelation or emotional intuition? Can we be certain about our knowledge – or must we always remain in doubt?

Does the candidate demonstrate an intuitive pathway to his/her knowledge? Does the candidate work with a mind map of complexity tradeoff, loop invariant, problem size variation, back checking, quick calculation etc.

For example, consider this case: Candidate does very well till score 100 (two ball case) but cannot do the three ball case (20 bonus points). What does this mean? Does the candidate know the problem, but s/he cannot tackle a simple variation?

Applying Ayn Rand to Tech Interview

How do we know what the candidate knows?

Remember that the narrative must feel close to heart – the whole process should appeal to the interviewer. The authenticity of the interaction and engagement should stand out.

Where is this Going?

So far, this article has been about how to think about the algorithm (making progress while maintaining invariant). Let us now strip all algorithmic thinking from this article. What else remains? This is what we think this article has besides algorithm:

- Structuring Interview
- Problem Design
- Having an objective philosophy (abstract architecture of engagement)

This is my point besides algorithms or beyond technical:

Conducting a good technical interview is not easy!

Thought leadership needs to come from the interviewer to create a platform or high potential for interviewer to perform. This creativity from interviewers is needed since we are engaging in human interaction.

Other Interview Problems

Before looking for other similar problems, let us look at the design of this problem:

- Trade off space complexity with time complexity
- Increasing level of sophistication
- Natural (not contrived) description

A problem with space-time tradeoff is by definition computationally elegant (evidence is Turing Machine or Computer Architecture). Increasing the level of sophistication is needed for structuring interview. A problem described in terms of physical setting (100 story building, dropping balls) requires the candidate to model the problem and analyze the solution in that model space. So if you look for another interview problem, please keep these three attributes in mind.

Conclusion

We will conclude by going back to introduction. We said, the success depends on three factors:

- Candidate's preparedness (technical and communication)
- Interviewer's preparedness (interviewing and communication)
- Luck (both on the part of candidate and interviewer)

Hopefully, we could articulate on item 1 and 2 above. We all need luck since several things need to line up in order to make a good hire in today's competitive job market for software engineers. Good luck with your interview!

References

- Blog Post: Interview Questions: Two Bowling Balls. Jesse Farmer on Tuesday, April 15, 2008, <http://20bits.com/article/interview-questions-two-bowling-balls>, This blog post has a different take on the same problem.
- Book: How to Think About Algorithms, Professor Jeff Edmonds, <http://www.cambridge.org/us/academic/subjects/computer-science/algorithms-complexity-computer-algebra-and-computational-g/how-think-about-algorithms>
- There are many algorithm texts that provide lots of well-polished code and proofs of correctness. This book is not one of them. Instead, this book presents insights, notations, and analogies to help the novice describe and think about algorithms like an expert. By looking at both the big picture and easy step-by-step methods for developing algorithms, the author helps students avoid the common pitfalls. He stresses paradigms such as loop invariants and recursion to unify a huge range of algorithms into a few meta-algorithms. Part of the goal is to teach the students to think abstractly. Without getting bogged with formal proofs, the book fosters a deeper understanding of how and why each algorithm works. These insights are presented in a slow and clear manner accessible to second- or third-year students of computer science, preparing them to find their own innovative ways to solve problems.
- Book: Introduction to objective epistemology, Ayn Rand, http://en.wikipedia.org/wiki/Introduction_to_Objectivist_Epistemology
- The majority of the book is Rand's summation of the Objectivist theory of concepts and solution to the problem of universals. An additional essay by Peikoff discusses the analytic-synthetic distinction. A second edition published in 1990 includes transcripts of a discussion session Rand conducted on epistemology.

About the Authors

Soumen Sarkar is a Senior Technical Product Manager in Platform team of WalmartLabs where he manages several products based on frameworks that operate in a high-scale, distributed, multi-tenancy private cloud environment. Before WalMartLabs, Soumen worked in high scale platforms with Akamai, Nokia, Yahoo and eBay. He graduated from Indian Institute of Technology (IIT) with a masters in Electrical Engineering. E-mail: soumen.sarkar@gmail.com.

Professor Jeff Edmonds is at the Theory Group of Department of Computer Science & Engineering, York University. Jeff Edmonds received his Ph.D. in 1992 at the University of Toronto in theoretical computer science. His thesis proved that certain computation problems require a given amount of time and space. He did his post doctorate work at the ICSI in Berkeley on secure multi-media data transmission and in 1995 became an Associate Professor in the Department of Computer Science at York University, Canada. He has worked extensively at IIT Mumbai, India, and University of California San Diego. He is well published in the top theoretical computer science journals on topics including complexity theory, scheduling, proof systems, probability theory, combinatorics, and, of course, algorithms.

A Natural Programming Method.

Programming with Natural Language

by Tsun-Huai Soo from Taiwan

Are English and C different languages? Yes, the former is a natural language, and the latter is a computer language. Do they share something in common? Yes, they do, they are both logical languages. Since they are both logical languages, why don't we just write English text to run a computer? Why do we program a computer with a computer language rather than a natural language? In the following text we will be using natural language to program. When we refer to characters, we mean Hanji, aka Kanji in Japanese or Hanja in Korean, and Kana, Hangul. Characters are romanized when present in following text. One cell of a lookup table holds a definition for the table key. A definition is a language element such as a statement, a declaration, and so on.

Below is a dictionary for terms we use in the following paragraphs.

Term	Meaning
Hangul	syllabic blocks of letters as Korean characters
Hanja	Chinese characters
Hanji	Chinese characters
Japanese language	a language using Kanji, Hirakana, and Katakana
Kana	syllabic Japanese characters known as Hirakana and Katakana
Kanji	Chinese characters
Korean language	a language using Hangul and Hanja
Taiwanese language	a language using Hanji and Latin alphabet

Below is another dictionary for the characters we use in the following paragraphs.

Character	Meaning
ÌN	v. to print
LIÓK	v. to record, n. records
Character	Meaning
SOK	v. to speed up
TSUÁN	v. to transfer, n. transfers

Programming with Natural Language

Let's have a look at the example code before diving into the details. Below is a lookup table:

```
LIÓ'K: def record end
      ÌN: puts "a short example"; puts "This is a longer example."
```

The table has two Hanji table keys. For one of the table keys LIÓK, which means “to record”, it has one definition “def exemplify end.” For the other table key ÌN, which means to print, it has two definitions. One is puts “a short example,” the other is puts “This is a longer example.” They are separated by a semicolon. We can then compose LIÓK and ÌN to form a phrase “LIÓ K ÌN.” Any two given characters can be combined together and remain syntactically correct. No declension or conjugation is required. The pronunciation does change but we don't cover this topic here. Below is the combined phrase and its correspondent code snippet:

```
LIO' K ÌN
def record
  puts "This is a longer example"
end
```

Why is only one definition of ÌN is combined with LIO'K's definition? The selection of one of the definitions is regulated by a lookup rule as shown below:

```
LIO'K.ÌN:0.2
```

The rule says, when ÌN is appended after LIO'K, the 2nd definition of ÌN is selected and 0 means “don’t care” for LIO'K. And why is ÌN nested in LIO'K? Because the combination rule for LIO'K and ÌN is as following:

```
LIO'K > ÌN
```

A Greater Than symbol can mean the 1st character nest the 2nd one.

The Ambiguity

A character can have more than one definition when it is composed in a sentence. The same rule holds true for an English word. The above code snippets are an unambiguous example. Here is an ambiguous English example:

```
The house can be bought for 10 dollars or 1 million dollars.
```

The subject “the house” has 2 meanings. For 10 dollars it means a toy house. For 1 million dollars it means a concrete house. The lookup table for this English sentence is shown as below:

```
The house: acquire(theToyHouse); acquire(theConcreteHouse);
10 dollars: pay(10);
1 million dollars: pay(1000000);
can be bought for: void buy() {}
```

The lookup rule is as following:

```
The house.10 dollars:1.0
The house.1 million dollars:2.0
```

The first definition of “The house” will be selected for 10 dollars. The second definition of “The house” will be selected for 1 million dollars. We would then have 2 meanings for this sentence. One is “The toy house can be bought for 10 dollars.” The other is “The concrete house can be bought for 1 million dollars.” The combination rules is as following:

```
void buy () {
  acquire(theToyHouse);
  pay(10);
  acquire(theConcreteHouse);
  pay(1000000);
}
```

“can be bought for” will nest all other statements, which is presented by a Greater Than symbol. There are no combination rules set for others words in the sentence, so sequential combination is assumed for the other 4 statements:

```
can be bought for >
```

Not Polymorphic

Polymorphism is a feature of object-oriented programming. Polymorphism allows an object of a derived class to be passed to a polymorphic function which accepts a reference to its base class. The is-a or kind-of relationship is guaranteed through inheritance of class. Thus we say the derived class is substitutable for base class.

The method has nothing to do with polymorphism. It does compose polymorphic code, but it is not involved in dynamic binding. When program code is composed, we are also composing a natural phrase or sentence in parallel. We may as well use the literal meaning of is-a or kind-of relationship to refer to the polysemy or monosemy of a word or character. The relationship is then obviously not guaranteed through inheritance of class, but through polysemy or monosemy of a word or character.

Another Perspective of Code

The method is not intended to replace computer language. In fact, it provides a new perspective of source code.

When we write program in a traditional way, firstly a thought or idea is formed in our brain. We then break down the thought or idea into logical units. The process of breaking down the thought or idea is mostly conducted by induction and deduction. The logical units, such as function calls, variables, or class declarations, are constructed and organized into a working program. Then this well composed program code is ready to be parsed, interpreted or compiled, and then executed.

From another perspective of source code, we can break it up into statements. Each statement in a program code is composed in such a way that it forms a nested or sequential combination with previous or next statement. Given that, each statement can be represented by a word or character. We try to make it easier to understand by assigning only one word or character to each statement, a statement could be represented by a couple of words or characters though. We can then have a sequence of words or characters, which should be in itself a phrase or sentence of natural language.

Not a Mere Translation

A computer language is made of a fixed set of keywords and punctuations. Furthermore the keywords of main stream computer languages are all English words. There have been efforts made to develop non-English-based computer languages. They are, however, not widely used all over the world. The non-English-based computer languages have non-English keywords. They are suitable for those programmers whose mother tongues are not English.

Do English-based and non-English-based computer languages have something in common? Yes, they are all logical languages. A few of them are mere translations from existing computer languages. By translation I mean the keywords are translated from English-based computer languages.

The method is not a mere translation from any computer languages. As the title suggests, a programming method has to be operable. We can operate the method to write computer programs. In other words, we write computer programs via this method.

The method is also not a Text Entry Method/Input Method. A text entry method is just a software tool we use to generate a sequence of words or characters by typing on keyboard. The generated sequence of words or characters will then be used by the method. The method can sure co-work with non-English-based computer languages.

Reducing Barriers to Entry

The method is a step further from computer language to human beings. It intermediates between computer language and programmers. Since it intermediates, there has to be a mechanism for it. The words or phrases we type in natural language will be mapped to certain statements in computer language via this mechanism. Since we type in natural language, we understand the source code through natural lexicon and sentences.

Since we can understand computer language through natural language, we definitely reduce the entry barriers to computer languages.

When we assign a word or character to a statement, the word or character itself can be regarded as a comment to the statement. It may not be an elaborate one, but it could just co-exist with source code comments. They can refer to each other.

Adaptive Design

When we assign a set of statements to a word or character, what we are actually doing is providing options to it. One of the provided options will be selected according to the context of a natural language phrase or sentence. In other words, the action of selecting means adapting to the context. If the context changes, the selection changes. When code can be adaptive to the context, it can be more flexible and responsive to the context.

We can view source code as 2 parts in terms of adaptation, one part is adaptive and the other is unadaptive. Where does adaptation come from? A software program may be fitted into a variety of hardware. A software program may be even used by different users under different conditions. An adaptive software program can be responsive to the context with the support of its unadaptive program core. The adaptive part of software program makes the unadaptive part flexible and responsive, and the unadaptive part of software program provides core functionality to the adaptive part and support it.

Let's add a new entry TSUÁN to the lookup table:

```
LIO' K: def record end
TSUÁN: def transfer end
      ÌN: puts "a short example"; puts "This is a longer example."
```

Add a lookup rule like this:

```
TSUÁN.ÌN:0.1
```

And add a combination rule like this:

```
TSUÁN > ÌN
```

We can hence get the code snippet by writing a phrase TSUÁN ÌN:

```
TSUÁN ÌN
def transfer
  puts "a short example"
end
```

It is obvious that we switch the meaning of ÌN by replacing LIÓK with TSUÁN, so that ÌN is adaptive and responsive to the changing context of ÌN.

We can also add unadaptive code to the snippet. Provided the character SOK means to speed up the string-printing routine "puts", we can add a new entry SOK to the lookup table:

```
LIO'K: def record end
TSUÁN: def transfer end
      SOK: speedUp
      ÌN: puts "a short example"; puts "This is a longer example."
```

And add 2 entries of combination rules:

```
LIO' K > SOK
TSUÁN > SOK
```

Hence for the 2 phrases LIO' K SOK ÌN and TSUÁN SOK ÌN, we have their relative code snippets:

```
LIO' K SOK ÌN
def record
  speedUp
  puts "This is a longer example."
end
```

and

```
TSUÁN SOK ÌN
def transfer speedUp
  puts "a short example"
end
```

SOK is therefore the unadaptive part of the code.

Co-working with Existing Computer Languages

The method doesn't replace existing computer language. Natural language will be co- working with computer language. The operations applied on natural languages will be reflected on the combinations of statements of computer languages. The following are examples for the method to co-work with different programming languages, such as *the lookup table for Python*:

```
LIO' K: def record(): TSUÁN: def transfer():
SOK: speedUp();
ÌN: print 'python code'; print 'longer python code'
```

The Python code snippet for the phrase TSUÁN SOK ÌN:

```
TSUÁN SOK ÌN
def transfer():
  speedUp();
  print 'python code'
```

The lookup table for C:

```
LIO' K: void record(){}
TSUÁN: void transfer(){}
SOK: speedUp();
ÌN: printf("c code\n"); printf("longer c code\n");
```

The C code snippet for the phrase TSUÁN SOK ÌN:

```
TSUÁN SOK ÌN
void transfer(){
  speedUp();
  printf("c code");
}
```

The lookup table for Swift:

```
LIO' K: func record() {}
TSUÁN: func transfer() {}
SOK: speedUp()
ÌN: println("swift code"); println("longer swift code")
```

The Swift code snippet for the phrase TSUÁN SOK ÌN:

```
TSUÁN SOK ÌN
func transfer(){
    speedUp()
    println("swift code")
}
```

The lookup table for XML Markup languages:

```
LIO' K: <records> </records>
TSUÁN: <transfers> </transfers>
SOK: speedy xml text.
ÌN: xml text; longer xml text
```

The XML code snippet for the phrase TSUÁN SOK ÌN:

```
TSUÁN SOK ÌN
<transfer>
    speedy xml
    text. xml text
</transfer>
```

Not a Programming Paradigm

The method can be applied to existing programming paradigms, being it functional, procedural, or object-oriented. Given a code snippet, it can be parsed into 2 types of combination. One type is nested combination, and the other type is sequential combination. In the example of C language, a for-loop can enclose a couple of statements. We can say for- loop is the nesting statement and the enclosed statements are the nested statements. In the example of a function definition, the function name and its return type are the nesting statement and the function body is the nested statement. The next one is sequential combination. When 2 statements are aligned with the same indentation, we say they form a sequential combination. For example, when 2 function calls are aligned, we say they are sequential.

The method is not a new programming paradigm. In other words, a programming method is not a programming paradigm. The method can also be applied to any newly invented paradigms.

The lookup table for Object-oriented paradigm:

```
LIO' K: void record (PrintingObject prnObj) {}
TSUÁN: void transfer (PrintingObject prnObj) {}
SOK: speedUp();
ÌN: prnObj.Print(); prnObj.PrintLonger();
```

The Object-Oriented code snippet for the phrase TSUÁN SOK ÌN:

```
TSUÁN SOK ÌN
void transfer(PrintingObject prnObj){
    speedUp();
    prnObj.Print();
}
```

The lookup table for Scripting paradigm:

```
LIO' K: function record(){} TSUÁN: function transfer(){}
SOK: speedUp();
ÌN: alert('javascript code'); alert('longer javascript code')
```

The Scripting code snippet for the phrase TSUÁN SOK ÌN:

```
TSUÁN SOK ÌN
void transfer(){
    speedUp();
    alert('javascript code');
}
```

The lookup table for Functional paradigm:

```
LIO' K: (defun record ())
TSUÁN: (defun transfer())
SOK: (funcall speedup 10)
ÌN: (format t "lisp code"); (format t "longer lisp code")
```

The Functional code snippet for the phrase TSUÁN SOK ÌN:

```
TSUÁN SOK ÌN
(defun transfer()
  (funcall speedup 10)
  (format t "lisp code")
)
```

The lookup table for Declarative paradigm:

```
LIO' K: #records {}
TSUÁN: #transfers {}
SOK: transition: width 10s;
ÌN: background-color: #ffffff; background-color: #000000
```

The Declarative code snippet for the phrase TSUÁN SOK ÌN:

```
TSUÁN SOK ÌN
#transfers {
  transition: width 10s;
  background-color: #ffffff
}
```

Software Developer's

new ideas & solutions for professional programmers

JOURNAL