Journal of Software Engineering and Applications, 2014, 7, 670-678
Published Online July 2014 in SciRes. http://www.scirp.org/journal/jsea
http://dx.doi.org/10.4236/jsea.2014.78061

Software Frameworks, Architectural and Design Patterns

Njeru Mwendi Edwin

Department of Computing, Jomo Kenyatta University of Agricluture & Technology, Nairobi, Kenya
Email: nmwendi@gmail.com

Received 21 April 2014; revised 20 May 2014; accepted 14 June 2014

Abstract

Software systems can be among the most complex constructions in engineering disciplines and can span into years of development. Most software systems though implement in part what has already been built and tend to follow known or nearly known architectures. Although most software systems are not of the size of say Microsoft Windows 8, complexity of software development can be quick to increase. Thus among these methods that are the most important is the use of architectural and design patterns and software frameworks. Patterns provide known solutions to re-occurring problems that developers are facing. By using well-known patterns reusable components can be built in frameworks. Software frameworks provide developers with powerful tools to develop more flexible and less error-prone applications in a more effective way. Software frameworks often help expedite the development process by providing necessary functionality "out of the box". Providing frameworks for reusability and separation of concerns is key to software development today. In this study I take a look at the state of art and the impact of frameworks and patterns in software development.

Keywords

Software Frameworks, Architectural Patterns, Design Patterns

1. Introduction

A framework is an integrated collection of components that collaborate to produce a reusable architecture for a family of related applications. Design patterns represent solutions to problems that arise when developing software within a particular context, capture the static and dynamic structure and collaboration in software designs thus facilitate reuse of successful software architectures and designs. Software systems can be very complex constructions and can span into years of development [1]. Overtime software engineering processes have sought to *reduce time to market*, *reduce the cost of development*, *standardize software development*, *improve quality*, *improve reliability* and *reduce the complexity in process management*. Most software systems though implement in part what has already been built and tend to follow known or nearly known architectures. It is notable that complexity of software development can be quick to increase due to its nature such as flexibility, extensibility, hidden constraints and the lack of visualization to the owners. Thus among these methods that are the most important in re-use of the known is the use of architectural and design patterns and software frameworks. It's notable though frameworks are usually domain-specific and applicable only to families of applications [2].

2. Software Frameworks

A software framework is a concrete or conceptual platform where common code with generic functionality can be selectively specialized or overridden by developers or users. Frameworks take the form of libraries, where a well-defined application program interface (API) is reusable anywhere within the software under development. [3].

Certain features make a framework different from other library forms, including the following:

- *Default Behavior*: Before customization, a framework behaves in a manner specific to the user's action.
- *Inversion of Control*: Unlike other libraries, the global flow of control within a framework is employed by the framework rather than the caller.
- *Extensibility*: A user can extend the framework by selectively replacing default code with user code.
- *Non-modifiable Framework Code*: A user can extend the framework but not modify the code. The purpose of software framework is to simplify the development environment, allowing developers to dedicate their efforts to the project requirements, rather than dealing with the framework's mundane, repetitive functions and libraries [3].

For example, rather than creating a VoIP application from scratch, a developer using a prepared framework can concentrate on adding user-friendly buttons and menus, or integrating VoIP with other functions.

Software frameworks consist of *frozen spots* and *hot spots*. *Frozen spots* define the overall architecture of a software system, that is to say its basic components and the relationships between them. These remain unchanged (frozen) in any instantiation of the application framework. *Hot spots* represent those parts where the programmers using the framework add their own code to add the functionality specific to their own project.

In an object-oriented environment, a framework consists of abstract and concrete classes. Instantiation of such a framework consists of composing and subclassing the existing classes.

When developing a concrete software system with a software framework, developers utilize the hot spots according to the specific needs and requirements of the system. Software frameworks rely on the Hollywood Principle: "Don't call us, we'll call you". This means that the user-defined classes (for example, new subclasses), receive messages from the predefined framework classes. Developers usually handle this by implementing superclass abstract methods.

While frameworks generally refer to broad software development platforms, the term can also be used to describe a specific framework within a larger programming environment. For example, multiple Java frameworks, such as Spring, ZK, and the Java Collections Framework (JCF) can be used to create Java programs. Additionally, Apple has created several specific frameworks that can be accessed by OS X programs. These frameworks are saved with a .FRAMEWORK file extension and are installed in the /System/Library/Frameworks directory. Examples of OS X frameworks include AddressBook.framework, CoreAudio.framework, CoreText.framework, and QuickTime.framework [4].

There are frameworks that cover specific areas of application development such as JavaScript/CSS frameworks that target the presentation (view) layer of the application, and there are others that handle more of the dynamic aspects of the application. Some include both!

Examples of frameworks that are currently used or offered by standards bodies or companies include:

- *Resource Description Framework*, a set of rules from the World Wide Web Consortium for how to describe any Internet resource such as a Web site and its content.
- *Internet Business Framework*, a group of programs that form the technological basis for the mySAP product from SAP, the German company that markets an enterprise resource management line of products.
- *Sender Policy Framework*, a defined approach and programming for making e-mail more secure.
- *Zachman framework*, a logical structure intended to provide a comprehensive representation of an informa-

tion technology enterprise that is independent of the tools and methods used in any particular IT business [5].

Examples of frameworks used in development today include: Zend framework for PHP, Spring framework for Java, .NET Framework (ASP.NET MVC), Django for python, Java Server Faces, Java apache cocoon etc.

Advantages [2]

- Reuse code that has been pre-built and pre-tested. Increase the reliability of the new application and reduce the programming and testing effort, and time to market.
- A *framework* can help establish better programming practices and appropriate use of design patterns and new programming tools.
- A *framework* can provide new functionality, improved performance, or improved quality without additional programming by the framework user.
- By *definition*, a framework provides you with the means to extend its behaviour.

Disadvantages [2]

- *Creating* a framework is difficult and time-consuming (*i.e.* expensive).
- The *learning* curve for a new framework can be steep.
- Over *time*, a framework can become increasingly complex.
- *Frameworks* often add to the size of programs, a phenomenon termed "code bloat".

3. Architectural and Design Patterns

In software engineering, a design pattern is a general reusable solution to a commonly occurring problem in software design. A design pattern is not a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations. Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved (wikibooks).

Design patterns reside in the domain of modules and interconnections. At a higher level there are architectural patterns that are larger in scope, usually describing an overall pattern followed by an entire system.

There are many types of design patterns:

Structural patterns address concerns related to the high level structure of an application being developed.

Computational patterns address concerns related to the identification of key computations.

Algorithm strategy patterns address concerns related to high level strategies that describe how to exploit application characteristic on a computation platform.

Implementation strategy patterns address concerns related to the realization of the source code to support how the program itself is organized and the common data structures specific to parallel programming.

Execution patterns address concerns related to the support of the execution of an application, including the strategies in executing streams of tasks and building blocks to support the synchronization between tasks.

Design patterns can speed up the development process by providing tested, proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation. Reusing design patterns helps to prevent subtle issues that can cause major problems, and it also improves code readability for coders and architects who are familiar with the patterns. In addition to this, patterns allow developers to communicate using well-known, well understood names for software interactions.

In order to achieve flexibility, design patterns usually introduce additional levels of indirection, which in some cases may *complicate the resulting designs* and *hurt application performance*.

By definition, a pattern must be programmed a new into each application that uses it [2] [6].

4. Classification and List of Patterns

Design patterns were originally grouped into the categories: *creational patterns*, *structural patterns*, and *behavioral patterns*, and described using the concepts of *delegation*, *aggregation*, and *consultation* [6] [7].

In Design Patterns, an *aggregate* is not a design pattern but rather refers to an object such as a list, vector, or generator which provides an interface for creating iterators [6].

In software engineering, *delegation* can be thought of as passing execution to another object while retaining the identity of the calling object. Thus, if the object delegated to calls another method of its "self", that self is the original object, not the object delegated to. If, on the other hand, we hand off processing to another object, and the object acts on its own, with its own "self", and no implicit reference to the original message receiver, that is

Consultation.
Examples:

4.1. Creational Patterns [6]

- *Abstract factory*: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- *Builder*: Separate the construction of a complex object from its representation allowing the same construction process to create various representations.
- *Factory method*: Define an interface for creating an object, but let subclasses decide which class to instantiate. *Factory* Method lets a class defer instantiation to subclasses.
- *Lazy initialization*: Tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed.
- *Multiton*: Ensure a class has only named instances, and provide global point of access to them.
- *Object pool*: Avoid expensive acquisition and release of resources by recycling objects that are no longer in use. Can be considered a generalization of connection pool and thread pool patterns.
- *Prototype*: Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- *Resource acquisition is initialization*: Ensure that resources are properly released by tying them to the lifespan of suitable objects.
- *Singleton*: Ensure a class has only one instance, and provide a global point of access to it.

4.2. Structural Patterns

- *Adapter or Wrapper*: Convert the interface of a class into another interface clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces.
- *Bridge*: Decouple an abstraction from its implementation allowing the two to vary independently.
- *Composite*: *Compose* objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- *Decorator*: Attach *additional* responsibilities to an object dynamically keeping the same interface. Decorators provide a flexible alternative to subclassing for extending functionality.
- *Facade*: Provide a *unified* interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- *Front Controller*: Provide a unified interface to a set of interfaces in a subsystem. Front Controller defines a higher-level interface that makes the subsystem easier to use.
- *Flyweight*: Use sharing to *support* large numbers of fine-grained objects efficiently.
- *Proxy*: Provide a surrogate *or* placeholder for another object to control access to it.

4.3. Behavioral Patterns

- *Blackboard*: Generalized observer, which allows multiple readers and writers. Communicates information system-*wide*.
- *Chain of responsibility*: Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
- *Command*: Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- *Interpreter*: *Given* a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
- *Iterator*: Provide a way to access the elements of an aggregate object sequentially without exposing its underlying *representation*.
- *Mediator*: Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
- *Memento*: Without violating encapsulation, capture and externalize an object’s internal state allowing the

object to be restored to this state later.
- *Null object*: Avoid null *references* by providing a default object.
- *Observer or Publish/subscribe*: Define a *one*-to-many dependency between objects where a state change in one object results with all its *dependents* being notified and updated automatically.
- *Servant*: Define common *functionality* for a group of classes.
- *Specification*: Recombinable *business* logic in a Boolean fashion.
- *State*: Allow an object to *alter* its behavior when its internal state changes. The object will appear to change its class.
- *Strategy*: Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- *Template method*: Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- *Visitor*: Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

4.4. Concurrency Patterns

- *Active Object*: Decouples method execution from method invocation that reside in their own thread of control. The goal is to introduce concurrency, by using asynchronous method invocation and a scheduler for handling requests.
- *Balking*: Only execute an action on an object when the object is in a particular state.
- *Binding Properties*: Combining multiple observers to force properties in different objects to be synchronized or coordinated in some way.
- *Messaging pattern*: The messaging design pattern (MDP) allows the interchange of information (*i.e.* messages) between components and applications.
- *Double-checked locking*: Reduce the overhead of acquiring a lock by first testing the locking criterion (the "lock hint") in an unsafe manner; only if that succeeds does the actual lock proceed. Can be unsafe when implemented in some language/hardware combinations. It can therefore sometimes be considered an anti-pattern.
- *Event-based asynchronous*: Addresses problems with the Asynchronous pattern that occur in multithreaded programs.
- *Guarded suspension*: Manages operations that require both a lock to be acquired and a precondition to be satisfied before the operation can be executed.
- *Lock*: One thread puts a "lock" on a resource, preventing other threads from accessing or modifying it.
- *Monitor object*: An object whose methods are subject to mutual exclusion, thus preventing multiple objects from erroneously trying to use it at the same time.
- *Reactor*: A reactor object provides an asynchronous interface to resources that must be handled synchronously.
- *Read-write lock*: Allows concurrent read access to an object but requires exclusive access for write operations.
- *Scheduler*: Explicitly control when threads may execute single-threaded code.
- *Thread pool*: A number of threads are created to perform a number of tasks, which are usually organized in a queue. Typically, there are many more tasks than threads. Can be considered a special case of the object pool pattern.
- Thread-specific storage: Static or "global" memory local to a thread.

4.5. Data Access Patterns

Another interesting area where patterns have a wide application is the area of *data access patterns*.
- *ORM Patterns*: Domain Object Factory, Object/Relational Map, Update Factory.
- *Resource Management Patterns*: Resource Pool, Resource Timer, Retryer, Paging Iterator.
- *Cache Patterns*: Cache Accessor, Demand Cache, Primed Cache, Cache Collector, Cache Replicator.

- *Concurrency Patterns*: Transaction, Optimistic Lock, Pessimistic Lock.

4.6. Enterprise Patterns

If you deal with J2EE or with .Net Enterprise applications, the problems that occur and the solutions to them are similar. These solutions are the Enterprise patterns.

- *Presentation Tier Patterns*: Intercepting Filter, Front Controller, View Helper, Composite View, Service to Worker, Dispatcher View.
- *Business Tier Patterns*: Business Delegate, Value Object, Session Facade, Composite Entity, Value Object Assembler, Value List Handler, Service Locator.
- *Integration Tier Patterns*: Data Access Object, Service Activator.

4.7. Real-Time Patterns

Finally, in the area of real-time and embedded software development a vast number of patterns have been identified.

- *Architecture Patterns*: Layered Pattern, Channel Architecture Pattern, Component-Based Architecture, Recursive Containment Pattern and Hierarchical Control Pattern, Microkernel Architecture Pattern, Virtual Machine Pattern.
- *Concurrency Patterns*: Message Queuing Pattern, Interrupt Pattern, Guarded Call Pattern, Rendezvous Pattern, Cyclic Executive Pattern, Round Robin Pattern.
- *Memory Patterns*: Static Allocation Pattern, Pool Allocation Pattern, Fixed Sized Buffer Pattern, Smart Pointer Pattern, Garbage Collection Pattern, Garbage Compactor Pattern.
- *Resource Patterns*: Critical Section Pattern, Priority Inheritance Pattern, Priority Ceiling Pattern, Simultaneous Locking Pattern, Ordered Locking Pattern.
- *Distribution Patterns*: Shared Memory Pattern, Remote Method Call Pattern, Observer Pattern, Data Bus Pattern, Proxy Pattern, Broker Pattern.
- *Safety and Reliability Patterns*: Monitor-Actuator Pattern, Sanity Check Pattern, Watchdog Pattern, Safety Executive Pattern, Protected Single Channel Pattern, Homogeneous Redundancy Pattern, Triple Modular Redundancy Pattern, Heterogeneous Redundancy Pattern [8].

5. Software Frameworks and Software Performance

There are several factors that impact software and applications performance. The following are some of the industry's factors that impact application performance [9].

5.1. Application Complexity

Application complexity is one of the biggest factors impacting application performance. Today's applications and services, particularly those delivered via the Web, are a mosaic of components sourced from multiple places: data center, cloud, third-party, ect. While the customer or employee looking at a browser window sees a single application, multiple moving parts must execute in the expected manner to deliver a great end-user experience. Maybe the Web server and app server are running fine, but if the database is faltering, user experience will suffer. As the saying goes, "The more moving parts, the more that can go wrong". Frameworks can add to this complexity while reducing the developers time to production. The interdependency in APIs, platforms and implementations within the frameworks directly affect application development complexity.

5.2. Application Design

One of the biggest factors that impacts application performance is design. Performance must be designed in. When applications are specified, performance goals need to be delineated along with the details of the environment the applications will run in. Often development is left out of this and applications are monitored, analyzed and "fixed" after they are released into production. The only way to prevent poor app performance is to expose your app development to the rigorous quality controls and processes early on in the application lifecycle—and actually fix them early in the cycle. This is a critical part in deciding the implementation phase. Frameworks make the implementation phase easy if studied well but they should be considered in the design stage and analyzed for impact on software systems performance. Architectural and design patterns can if not well implemented impact heavily on delivered applications performance.

5.3. Application Testing

Today's applications are often developed in components and include lots of interfaces and integrations. Plugins are assumed tested and most code is simulated without testing performance on real-world networks. Before applications are deployed, transport across today's highly distributed network architectures should be monitored and optimized. Insufficient testing of the application in the actual production environment and under varying conditions impacts performance. Developers and testers need to have a clear understanding of the non-functional performance criteria. Frameworks provide some built in tools for testing code, APIs and plugins but strict quality controls and thorough testing must be done to achieve desired performance index.

5.4. The Butterfly Effect

The environmental variants need to be minimized and closely monitored to prevent the anomalous events in a software implementation successful—it's the choices you make in how you put them together to support the multiple environments within IT. Frameworks are built for specific families of applications and thus may improve performance.

5.5. The Infrastructure and Components of the Application Service

Application performance is impacted by components used to deliver the service to the user-frameworks and design patterns included, the user's interfaces with the application, and the connectivity between these components. The variance and complexity is what makes the problem hard to solve, and often causes approaches to fail on given architectures. One of the most critical factors that affect application performance, and often the hardest to identify and track, are application dependencies—on supporting applications, as well as the underlying system and network components that connect them all together. With the advent of virtualized servers and networks, the complexity of the application delivery infrastructure has increased significantly, and so the challenge is finding an application performance monitoring solution that can automatically discover and monitor the network and server topologies for the entire application service.

5.6. The Dynamic IT Environment: Virtualization and the Cloud

Applications today are an intricate mesh of multi-tier software running on servers, networks, and storage. In addition, there is a good chance they are running on virtualized hardware that is shared with other applications. It is very challenging in this dynamic environment to understand what will impact your application performance as it requires intimate knowledge of your ever-changing application structure at any given moment. Many IT organizations are very advanced on the application side but unfortunately still struggle to move beyond managing applications via a silo approach to the different technology tiers—application, server, network, storage, etc.

5.7. Mobility

One of the biggest factors we see is the acceleration of mobility and IT consumerization, which will propel the ongoing shift in application architectures required to deliver the most dynamic, modern mobile end user interfaces.

Software frameworks are to be designed based on either *elegance* or just a *problem solver*. Software elegance implies clarity, conciseness, and little waste. For example, "elegance" to a code generating framework would imply the creation of code that is clean and comprehensible to a reasonably knowledgeable programmer (and which is therefore readily modifiable), versus one that merely generates correct code [9].

6. Analysis and Suggestions

There is software in every applicable area of real world. As software systems have grown in complexity ranging from needs, integrations, collaborations, globalizations and security, software engineering has to meet the same.

This can be achieved by all software construction methodologies but it's easier and cheaper to deliver using frameworks and design patterns.

Due to the complexity of their APIs, the intended reduction in overall development time may not be achieved due to the need to spend additional time learning to use the framework; this criticism is clearly valid when a special or new framework is first encountered by developers. If such a framework is not used in subsequent tasks, the time invested in learning the framework can cost more than purpose-written code familiar to the project's staff; many programmers keep copies of useful boilerplate for common needs. Thus frameworks are useful if their learning is re-used.

As a framework is an application that is complete except for the actual functionality, you plug in the functionality and you have an application, they are very useful to developers. Software systems architects and designers can use helps guide the process and even produce designs already used.

Frameworks though add to the size of programs, a phenomenon termed "code bloat". Due to customer demand driven applications needs, both competing and complementary frameworks sometimes end up in a product. Frameworks if incorrectly applied could lead to loss of performance due to cross reference and non useful calls to APIs functions.

Consider, say, a GUI framework. The framework contains everything you need to make an application. Indeed you can often trivially make a minimal application with very few lines of source that does absolutely nothing-but it does give you window management, sub-window management, menus, button bars, etc. That's the framework side of things. By adding your application functionality and "plugging it in" to the right places in the framework you turn this empty app that does nothing more than window management, etc. into a real, full-blown application. This is the core use of frameworks kind of provide an environment with skeleton for meat to be put on.

There are similar types of frameworks for web apps, for server-side apps, etc. In each case the framework provides the bulk of the tedious, repetitive code (hopefully) while you provide the actual problem domain functionality. This is the ideal. In reality, of course, the success of the framework is highly variable and will be growing rapidly as we further the component based development, iterative and incremental object oriented development techniques.

Why not traditional libraries then? When you invoke a traditional library, you are still in control: you make the library calls that you want to make, and deal with the consequences. A framework inverts the flow of control: you hand over to it, and wait for it to invoke the various call-back functions that you provide. You put your program's life in its hands. That has consequences: one of the most important ones is that, while your program can use as many libraries as it likes, it can only use—or, rather, be used by—one framework.

Frameworks are jealous. They don't share. But some frameworks allow importing plugins or libraries into their repository as may be required.

A design pattern is a description or template for how to solve a problem that can be used in many different situations. Design patterns provide the much needed re-use of a solution. it doesn't provide any code that can be used in application development.

There is such a brighter future for design patterns especially as software gets more standardized and as environments evolve into an ecosystem of frameworks, designs and components.

The key is to build frameworks that can generate only useful code based on user needs. Elegant frameworks can be useful tools in ensuring great performance. A framework can help establish better programming practices and appropriate use of design patterns and new programming tools. A framework can provide enhanced functionality, improved performance, or improved quality without additional programming by the framework user especially extensible frameworks.

7. Conclusions

Traditionally development was based on lines of code. Today it's based on time to delivery. Complex systems are easily delivered and processes managed better as tools evolve to aid. Increasingly, software organizations are facing simultaneous pressures (Riehle, 2000):

- to reduce time to market;
- to reduce the cost of the product;
- to improve the productivity of the organization;

- to increase the reliability of the product; and
- to increase the quality of the product.

These pressures have resulted into evolving software environments from integrated development environments, frameworks, design modules and open source web based platforms. Same pressures have made online groups like stack overflow most used sites by software engineering peers.

The turnkey for software development is extensible frameworks and design patterns. That means a platform that one can pick what they need from architecture, design, framework parts, generic codes and patch them up into "MY" environment reducing the bulk of non required code and effectively the size of completed code. There are other advantages with this as I can pick up, extend and just build my applications that suit the purposes of my development, my metrics and quality standards. That's the future of frameworks, "fluid" frameworks.

References

[1] Ragnarsson Andri Ólafur, A. (2014) Importance of Design Patterns and Frameworks for Software Development. Reykjavik University, Reykjavik.

[2] Riehle, D. (2000) Framework Design: A Role Modeling Approach. Ph.D. Thesis, ETH Zürich, Zürich, No. 13509. http://dirkriehle.com/computer-science/research

[3] Janssen, C. (n.d.) Software Framework. http://www.techopedia.com http://www.techopedia.com/definition/14384/software-framework

[4] Techterms (2013) Framework. http://www.techterms.com/ http://www.techterms.com/definition/framework

[5] Rouse, M. (2005) Framework. http://whatis.techtarget.com http://whatis.techtarget.com/definition/framework

[6] Gamma, H.J. (1994) Design Patterns: Elements of Reusable Object Oriented Software. Addison-Wesley, Sydney, Zurich, Urbana, Hawthorne.

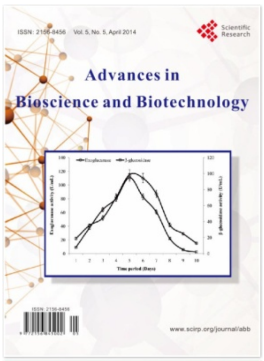
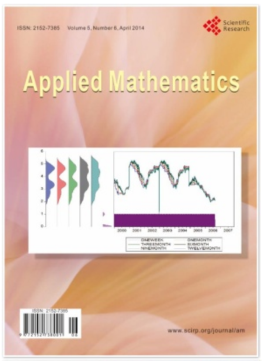
[7] Patterns, D. (2014) Design Patterns. http://www.oodesign.com/

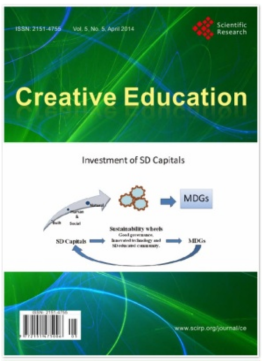
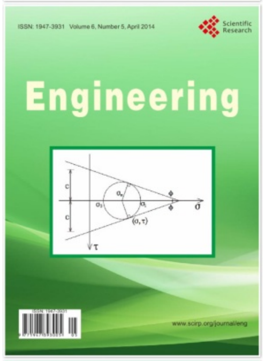
[8] Wikimedia (2013) Introduction to Software Engineering.

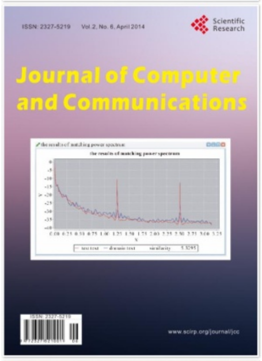
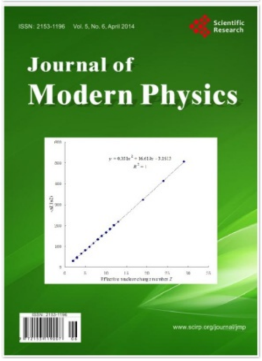
[9] Sanna, E.A. (2013) 15 Top Factors that Impact Application Performance. http://www.apmdigest.com http://apmdigest.com/15-top-factors-that-impact-application-performance

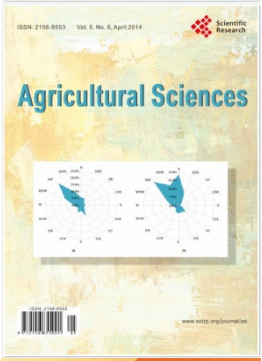
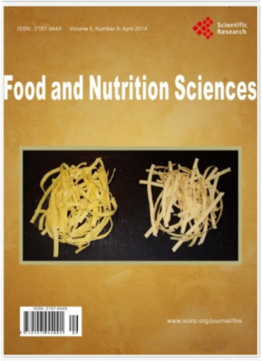
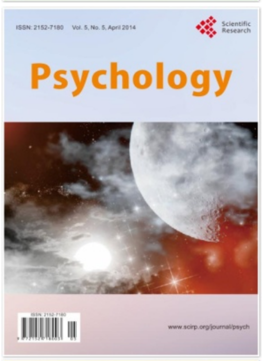
Glossary

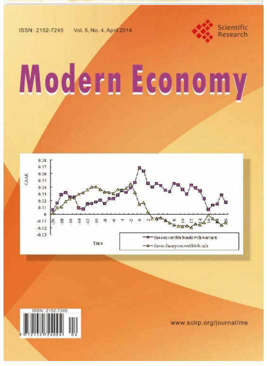
- API—An abbreviation of ***a**pplication **p**rogram **i**nterface*, is a set of routines, protocols, and tools for building software applications.
- VoIP—Voice over Internet Protocol is a category of hardware and software that enables people to use the Internet as the transmission medium for telephone calls by sending voice data in packets using IP rather than by traditional circuit transmissions of the PSTN.
- OS X frameworks—The OS X frameworks provide the interfaces you need to write software for Mac. Some of these frameworks contain simple sets of interfaces while others contain multiple sub-frameworks.
- ZK—Is an open-source Ajax Web application framework, written in Java, that enables creation of graphical user interfaces for Web applications with little required programming knowledge.
- CSS—**Cascading Style Sheets** (**CSS**) is a style sheet language used for describing the look and formatting of a document written in a markup language.
- GUI—Abbreviation for graphical user interface is a human-computer interface that uses windows, icons and menus and which can be manipulated by a mouse (and to a limited extent by a keyboard).
- IDE—Abbreviation for integrated development is a software application that provides comprehensive facilities to computer programmers for software development.


Advances in Bioscience and Biotechnology

Applied Mathematics

American Journal of Plant Sciences

Creative Education

Engineering

HEALTH

Journal of Computer and Communications

Journal of Modern Physics

Journal of Environmental Protection

Agricultural Sciences

Food and Nutrition Sciences

Psychology

Natural Science

Modern Economy

Journal of Cancer Therapy

American Journal of Analytical Chemistry