**Міністерство освіти і науки, молоді та спорту**

**Національний технічний університет України**

**“Київський політехнічний інститут”**

Факультет інформатики та обчислювальної техніки

Кафедра технічної кібернетики

**РЕФЕРАТ**

з дисципліни “Англійська мова професійного спрямування

на тему: “Software (program) design”

Виконала: Оляницька Єлизавета Олексіївна

студентка групи ІК-83

Викладач: Грабовська Н.A.

Київ 2021

**Contents**

1. Introduction………………………………………………………………….2
2. Design concepts…………………………………………………………..…3
3. Design considerations……………………………………………………….4
4. Design patterns…………………………………………………..………….5
5. Summary………………………………………………………...…………..7
6. Summary translation………………………………………………….…….7
7. References…………………………………………………………………..8
8. Glossary………………………………………………………….………….9

**Introduction**

Software or program design is the process of envisioning and defining software solutions to one or more sets of problems. One of the main components of software design is the software requirements analysis (SRA). SRA is a part of the software development process that lists specifications used in software engineering. If the software is "semi-automated" or user centered, software design may involve user experience design yielding a storyboard to help determine those specifications. If the software is completely automated (meaning no user or user interface), a software design may be as simple as a flow chart or text describing a planned sequence of events. There are also semi-standard methods like Unified Modeling Language and Fundamental modeling concepts. In either case, some documentation of the plan is usually the product of the design. Furthermore, a software design may be platform-independent or platform-specific, depending upon the availability of the technology used for the design.

The main difference between software analysis and design is that the output of a software analysis consists of smaller problems to solve. Additionally, the analysis should not be designed very differently across different team members or groups. In contrast, the design focuses on capabilities, and thus multiple designs for the same problem can and will exist. Depending on the environment, the design often varies, whether it is created from reliable frameworks or implemented with suitable design patterns. Design examples include operation systems, webpages, mobile devices or even the new cloud computing paradigm.

Software design is both a process and a model. The design process is a sequence of steps that enables the designer to describe all aspects of the software for building. Creative skill, past experience, a sense of what makes "good" software, and an overall commitment to quality are examples of critical success factors for a competent design. It is important to note, however, that the design process is not always a straightforward procedure; the design model can be compared to an architect's plans for a house. It begins by representing the totality of the thing that is to be built (e.g., a three-dimensional rendering of the house); slowly, the thing is refined to provide guidance for constructing each detail (e.g., the plumbing lay). Similarly, the design model that is created for software provides a variety of different views of the computer software. Basic design principles enable the software engineer to navigate the design process.

**Design concepts**

A design concept is the core idea driving the design of a product, explained via a collection of sketches, images, and a written statement. This helps the designers and, later, the developers stay on track throughout the creative process, ensuring they bring a product to market with value to target users. They are as follows:

*Abstraction* - Abstraction is the process or result of generalization by reducing the information content of a concept or an observable phenomenon, typically in order to retain only information which is relevant for a particular purpose. It is an act of Representing essential features without including the background details or explanations.

*Refinement* - It is the process of elaboration. A hierarchy is developed by decomposing a macroscopic statement of function in a step-wise fashion until programming language statements are reached. In each step, one or several instructions of a given program are decomposed into more detailed instructions. Abstraction and Refinement are complementary concepts.

*Modularity* - Software architecture is divided into components called modules.

*Software Architecture* - It refers to the overall structure of the software and the ways in which that structure provides conceptual integrity for a system. Good software architecture will yield a good return on investment with respect to the desired outcome of the project, e.g. in terms of performance, quality, schedule and cost.

*Control Hierarchy* - A program structure that represents the organization of a program component and implies a hierarchy of control.

*Structural Partitioning* - The program structure can be divided into both horizontally and vertically. Horizontal partitions define separate branches of modular hierarchy for each major program function. Vertical partitioning suggests that control and work should be distributed top down in the program structure.

*Data Structure* - It is a representation of the logical relationship among individual elements of data.

*Software Procedure* - It focuses on the processing of each module individually.

*Information Hiding* - Modules should be specified and designed so that information contained within a module is inaccessible to other modules that have no need for such information.

In his object model, Grady Booch mentions Abstraction, Encapsulation, Modularisation, and Hierarchy as fundamental software design principles. The acronym PHAME (Principles of Hierarchy, Abstraction, Modularisation, and Encapsulation) is somemes used to refer to these four fundamental principles.

**Design considerations**

There are many aspects to consider in the design of a piece of software. The importance of each consideration should reflect the goals and expectations that the software is being created to meet. Some of these aspects are:

*Compatibility* - The software is able to operate with other products that are designed for interoperability with another product. For example, a piece of software may be backward-compatible with an older version of itself.

*Extensibility* - New capabilities can be added to the software without major changes to the underlying architecture.

*Modularity* - the resulting software comprises well defined, independent components which leads to better maintainability. The components could be then implemented and tested in isolation before being integrated to form a desired software system. This allows division of work in a software development project.

*Fault-tolerance* - The software is resistant to and able to recover from component failure.

*Maintainability* - A measure of how easily bug fixes or functional modifications can be accomplished. High maintainability can be the product of modularity and extensibility.

*Reliability* *(Software durability)* - The software is able to perform a required function under stated conditions for a specified period of time.

*Reusability* - The ability to use some or all of the aspects of the preexisting software in other projects with little to no modification.

*Robustness* - The software is able to operate under stress or tolerate unpredictable or invalid input. For example, it can be designed with resilience to low memory conditions.

*Security* - The software is able to withstand and resist hostile acts and influences.

*Usability* - The software user interface must be usable for its target user/audience. Default values for the parameters must be chosen so that they are a good choice for the majority of the users.[6]

*Performance* - The software performs its tasks within a time-frame that is acceptable for the user, and does not require too much memory.

Portability - The software should be usable across a number of different conditions and environments.

Scalability - The software adapts well to increasing data or added features or number of users.

**Design patterns**

Design patterns represent the best practices used by experienced object-oriented software developers. Design patterns are solutions to general problems that software developers faced during software development. These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.

In 1994, four authors Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides published a book titled Design Patterns - Elements of Reusable Object-Oriented Software which initiated the concept of Design Pattern in Software development.

These authors are collectively known as Gang of Four (GOF). According to these authors design patterns are primarily based on the following principles of object orientated design.

* Program to an interface not an implementation
* Favor object composition over inheritance

As per the design pattern reference book Design Patterns - Elements of Reusable Object-Oriented Software, there are 23 design patterns which can be classified in three categories: Creational, Structural and Behavioral patterns.

*Creational design* *patterns* are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or in added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation.

Creational design patterns are composed of two dominant ideas. One is encapsulating knowledge about which concrete classes the system uses. Another is hiding how instances of these concrete classes are created and combined.

Five well-known design patterns that are parts of creational patterns are the

* Abstract factory pattern, which provides an interface for creating related or dependent objects without specifying the objects' concrete classes.
* Builder pattern, which separates the construction of a complex object from its representation so that the same construction process can create different representations.
* Factory method pattern, which allows a class to defer instantiation to subclasses.
* Prototype pattern, which specifies the kind of object to create using a prototypical instance, and creates new objects by cloning this prototype.
* Singleton pattern, which ensures that a class only has one instance, and provides a global point of access to it.

*Structural design* *patterns* are design patterns that ease the design by identifying a simple way to realize relationships among entities.

Examples of Structural Patterns include:

* Adapter pattern: 'adapts' one interface for a class into one that a client expects
* Aggregate pattern: a version of the Composite pattern with methods for aggregation of children
* Bridge pattern: decouple an abstraction from its implementation so that the two can vary independently
* Composite pattern: a tree structure of objects where every object has the same interface
* Decorator pattern: add additional functionality to an object at runtime where subclassing would result in an exponential rise of new classes
* Proxy pattern: a class functioning as an interface to another thing, etc

*Behavioral design patterns* are design patterns that identify common communication patterns among objects. By doing so, these patterns increase flexibility in carrying out communication.

Examples of this type of design pattern include:

* Blackboard design pattern: provides a computational framework for the design and implementation of systems that integrate large and diverse specialized modules, and implement complex, non-deterministic control strategies
* Chain of responsibility pattern: Command objects are handled or passed on to other objects by logic-containing processing objects
* Command pattern: Command objects encapsulate an action and its parameters
* "Externalize the stack": Turn a recursive function into an iterative one that uses a stack
* Interpreter pattern: Implement a specialized computer language to rapidly solve a specific set of problems

**Summary**

Software design documentation may be reviewed or presented to allow constraints, specifications and even requirements to be adjusted prior to computer programming. Redesign may occur after review of a programmed simulation or prototype. It is possible to design software in the process of programming, without a plan or requirement analysis, but for more complex projects this would not be considered feasible. A separate design prior to programming allows for multidisciplinary designers and subject-matter experts (SMEs) to collaborate with highly skilled programmers for software that is both useful and technically sound.

**Summary translation**

Документація з проектування програмного забезпечення може бути переглянута або представлена, щоб дозволити скоригувати обмеження, технічні характеристики і навіть вимоги перед програмуванням на комп’ютері. Редизайн може відбутися після перегляду запрограмованого моделювання або прототипу. Можна розробляти програмне забезпечення в процесі програмування без аналізу плану чи вимог, але для більш складних проектів це не вважатиметься здійсненним. Окремий дизайн до програмування дозволяє мультидисциплінарним дизайнерам та експертам з питань предметів (МСП) співпрацювати з висококваліфікованими програмістами для програмного забезпечення, яке є одночасно корисним і технічно обґрунтованим.

**References**

1. Ralph, P. and Wand, Y. (2009). A proposal for a formal definition of the design concept. In Lyytinen, K., Loucopoulos, P., Mylopoulos, J., and Robinson, W., editors, Design Requirements Workshop (LNBIP 14), pp. 103–136. Springer-Verlag, p. 109 doi:10.1007/978-3-540-92966-6\_6.
2. Freeman, Peter; David Hart (2004). "A Science of design for software-intensive systems". Communications of the ACM. 47 (8): 19–21 [20]. doi:10.1145/1012037.10120547. S2CID 14331332.
3. Davis, A:"201 Principles of Software Development", McGraw Hill, 1995.
4. Booch, Grady; et al. (2004). Object-Oriented Analysis and Design with Applications (3rd ed.). MA, USA: Addison Wesley. ISBN 0-201-89551-X. Retrieved 30 January 2015.
5. Suryanarayana, Girish (November 2014). Refactoring for Software Design Smells. Morgan Kaufmann. p. 258. ISBN 978-0128013977.
6. Carroll, John, ed. (1995). Scenario-Based Design: Envisioning Work and Technology in System Development. New York: John Wiley & Sons. ISBN 0471076597.
7. Bell, Michael (2008). "Introduction to Service-Oriented Modeling". Service-Oriented Modeling: Service Analysis, Design, and Architecture. Wiley & Sons. ISBN 978-0-470-14111-3.
8. Judith Bishop. "C# 3.0 Design Patterns: Use the Power of C# 3.0 to Solve Real-World Problems". C# Books from O'Reilly Media. Retrieved 2012-05-15. If you want to speed up the development of your .NET applications, you're ready for C# design patterns -- elegant, accepted and proven ways to tackle common programming problems.
9. Dijkstra, E. W. (1988). "On the cruelty of really teaching computing science". Retrieved 2014-01-10.
10. Knuth, Donald E. (1989). "Notes on the Errors of TeX" (PDF).
11. Ralph, P., and Wand, Y. A Proposal for a Formal Definition of the Design Concept. In, Lyytinen, K., Loucopoulos, P., Mylopoulos, J., and Robinson, W., (eds.), Design Requirements Engineering: A Ten-Year Perspective: Springer-Verlag, 2009, pp. 103-136
12. <https://www.unitedwaygt.org/>
13. <https://airfocus.com/>
14. https://www.tutorialspoint.com/

**Glossary**

1. Adaptive capacity – the ability to learn from and respond to changes in the internal and external environment.
2. Anti-oppression practice – a practice that acknowledges oppression in societies, economies, cultures, and groups, and takes action to remove or negate the influence of that oppression.
3. Best practice – a practice that has been shown, through a rigorous process of evaluation, to be effective in improving outcomes for a particular population.
4. Context – the broader organizational, social, economic, and political environment in which a program operates. This includes program relationships, capacity, and other factors outside of the control of the program. Some contextual factors include community needs and assets, community economic, social and demographic characteristics, governmental policy, and the presence of similar or complementary programs.
5. Diversity and inclusion – involves respecting and embracing diverse abilities, perspectives, identities and experiences. It includes the promotion of equity – welcoming, celebrating and valuing the contributions of all.
6. Emerging practice – a practice, based on a theory that is being tested through a process of evaluation, to determine whether it is effective in improving outcomes for a particular population.
7. Evaluation – the systematic assessment of an initiative, program or policy, including the collection and analysis of qualitative and/or quantitative data. It not only looks at whether the program is achieving its intended objectives, but also explores why or why not and the implications for the organization. It may also include exploration of unintended outcomes.
8. Evidence – information that is presented to support or counter an assertion. Evidence can range from examples that demonstrate a point to information resulting from rigorous evaluation or research. Evidence is used to support the selection and implementation of program design elements.
9. Fidelity – the extent to which a program adheres to its original model.
10. Grey literature – articles, documents and research that have not been peer reviewed and are produced outside of traditional academic and commercial sources.
11. Indicators – measurable qualities which track the progress of the program, initiative or activity toward the intended outcome(s), and help assess whether or not the intended outcome(s) have been achieved. They can be both qualitative and quantitative.
12. Initiative – steps, actions or a set of programs intended to start a change process. An initiative often involves a number of related programs aimed at resolving a specific issue.
13. Innovation – a novel solution to a social problem that is more effective, efficient, or sustainable than present solutions. This can include a completely new idea or a new way of applying an existing solution in a different context.
14. Inputs – a component of a program logic model. Inputs are the financial, human and other resources that go into a program.
15. Intervention – an action that is undertaken to produce some result. An intervention often refers to a program or initiative implemented to produce change among participants or a community.
16. Logic model – a systematic and visual way to present the relationships among the available resources to operate the program, the planned activities and the changes or results that the program is hoping to achieve. It provides a means for thinking through and recording what is expected or desired at different points in time in the ‘life’ of a program.
17. Measures of success – another term for indicators.
18. Monitoring – the systematic collection and review of data that indicates the extent to which a program is achieving its intended objectives.
19. Needs assessment – a type of evaluation that is conducted to determine the need for a program, the current mechanism for filling that need, determining what gaps exist and the most appropriate ways for filling those gaps.
20. Objective – a concise statement that describes the goal, intended aim or purpose of the program. It is often used interchangeably with goal or intended result.