DOWNLOAD PDF PROGRAM DESIGN METHODS

Chapter 1 : Design methods - Wikipedia

Software Design Methods and Tools from University of Colorado System. Since many software developers are compulsive coders, they have created software over the years to help them do their job. There are tools which make design and its associated.

Know how to choose between OO approach and Functional approach. Four components of a design Regardless which approach you take, your finished design must have four components. The architecture is the framework of the solution, often comprised of "design patterns. Class Diagram, Module hierarchy chart structure chart. When used within a subsystem "architecture" is a class diagram. Interface design describes how the elements of the design will communicate with each other. It describes the "protocols" or "contracts" that allow elements of the design to request services of each other. In OO design, this component is called Class Skeletons. In a large system design, it means a communication protocol. Procedural design describes how the functional aspects of the software will work. It describes the logic and control structures for the operations in the design. Data design describes the data structures which will store the information used in the solution. Usually each component has its own representation or notation, but sometimes they are combined. In a large system, this would include database design. Design Principles There are several principles which guide the design process. Three important ones are decomposition, abstraction, and information hiding. To subdivide a solution into separable components that are named and addressable. Decomposition is the primary tool for attacking large problems, by using a "Divide and conquer" strategy. It helps us several ways: Allows for distribution of effort via parallel development. Makes testing easier by isolating errors Promotes reuse. Localizes changes for easier maintenance. Makes the solution simpler to comprehend The disadvantage is the cost of integrating all the separate pieces. To focus on the significant or essential aspects of a situation or problem and ignore irrelevant details. It simplifies a design by hiding low level details and keeping the emphasis on the main features of the solution. It promotes implementation independence, meaning we can change the implementation low level without affecting the entire system at a high level. It simplifies our thinking by allowing us to focus on the problem domain and defer the details of the implementation until later. It reduces errors by reducing the number of details we need to hold in short-term memory. Abstraction and software evolution Information Hiding Closely related to abstraction, information hiding refers specifically to suppressing or keeping private the internal details of a program component. Other components that wish to access it must do so via the components public interface. Information hiding gives a software design more integrity by controlling access to the internals of a component. Thoughtless or sloppy design can defeat the purpose of information hiding.

Chapter 2 : Architectural Programming | WBDG Whole Building Design Guide

PROGRAM DESIGN METHODS (4 Credits) Learning Outcomes: On successful completion of this course, students will be able to: Explain program design method; Apply the process of program developing; Design the application using program design method; Demonstrate the use of program design method; Explain the object oriented design; Design the.

Many software development projects have been known to incur extensive and costly design errors. The most expansive errors are often introduced early in the development process. This underscores the need for better requirement definition and software de sign methodology. Software design is an important activity as it determines how the whole software development task would proceed including the system maintenance. The design of software is essentially a skill, but it usually requires a structure which w ill provide a guide or a methodology for this task. A methodology can be defined as the underlying principles and rules that govern a system. A method can be defined as a systematic procedure for a set of activities. Thus, from these definitions, a met hodology will encompass the methods used within the methodology. Different methodologies can support work in different phases of the system life cycle, for example, planning, analysis, design and programming, testing and implementation. Svoboda de veloped the idea of a methodology further by proposing that there should be at least four components: The conceptual model is needed to direct or guide the designers to the relevant aspects of the system. The set of procedure provides the designer a systematic and logical set of activities to begin the design task. The evaluation criteria provide an obje ctive measurement of the work done against some established standard or specifications. A software design methodology can be structured as comprising of the software design process component and the software design representation or diagrammatic component. The process component is based on the basic principles established in the methodolog y while the representation component is the "blueprint" from which the code for the software will be built. It should be noted, that in practice, the design methodology is often constrained by existing hardware configuration, the implementation language,the existing file and data structures and the existing company practices, all of which would limit the solution space available to develop the software. The evolution of each software design needs to be meticulously recorded or diagramed, including the basis for choices made, for future walk-throughs and maintenance. The Design Process Design is a formation of a plan of activities to accomplish a recognized need. The need may be well defined or ill defined. When needs are ill-defined, it is likely due to the fact that neither the need nor problem has been identified. The design process is a process of creative invention and definition, it involves synthesis and analysis, and thus, is difficult to summarize in a simple design formula. Design is an applied science. In a software design problem, a number of solutions exist. The designer each word "designer" can also refer to "designers" must plan and execute the design strategy taking into account certain established design practices. The designer often has to fall back on previous experience gained and has to study the existing software methodologies designed by others, to analyze their advantages and disadvantages. It is useful also to review the basic parameters, especially the requirements and system specifications. A designer must constantly improve and enrich his store of design solution. The development of design alternatives should be a regular design activity aimed at seeing the most rational solution. In the design of software, often there are different design methodologies that can be used to derive a software solut ion, this is called the design degree of freedom. A design solution can also have several degrees of freedom, which implies that there is possibly at least one solution in the solution space, often there is more than one solution. It must be noted that there are no unique answers for design. However, that does not imply that any answer will do. Some solutions are more optimal than others. A design is subject to certain problem-solving constraints, for example, in a vacation design problem, the const raints are money and time. Note also that there are constraints on the solutions, for example, users must be satisfied. Then in synthesis, it is necessary to begin with the solution of the main design problems and separate secondary items from the main ones. Successful design often begins with a clear definition of the design objectives. While in analysis and evaluation, the designer using these knowledge and simple diagrams, makes a first draft of the design in

the form of diagrams. This has to be t horoughly analyzed to verify that it meets the design objectives and that it is adequate but not over designed. A systematic approach is useful only to the extent that the designer is presented with a strategy that he can use as a base for planning the r equired design strategy for the problem at hand. A design problem is not a theoretical construct. Design has an authentic purpose - the creation of an end result by taking definite action or the creation of something having physical reality. The design process is by necessity an iterative one. The software design should describe a system that meet the requirements. It should be thorough, in-depth, complete and use standardized notations to depict the structure and systems. According to Ford , p. A good design is complete it will build everything required , economical it will not b uild what is not required , constructive it says how to build the product , uniform it uses the same building techniques throughout , and testable it can be shown to work. The Role of Design Methodology The role of the software design methodology cannot be overemphasized Freeman, Software design methodology provides a logical and systematic means of proceeding with the design process as well as a set of guidelines for decision-making. The desi gn methodology provides a sequence of activities, and often uses a set of notations or diagrams. The design methodology is especially important for large complex projects that involve programming-in-the-large where many designers are involved ; the use of a methodology establishes a set of common communication channels for translating design to codes and a set of common objectives. In addition, there must be an objective match between the overall character of the problem and the features of the solutio n approach, in order for a methodology to be efficient. Design Phase in Software Systems Development Pressman has described the software development process as consisting of three broad generic phases - the definition, development and maintenance phases. The definition phase defines the "what" of the software system, the development phase define s the "how" and the maintenance phase defines the support and future necessary changes. Almost every text on software development includes a SDLC model, there are some variations but, in general, the basic phases or activities are always present. The basic phases that are ever present are the analysis, design, testing, implementation and maintenance phases. The analysis phase involves the requirement definition, from which the software specifications are derived. Design then translates the specification into a set of notations that represents the algorithms, data structures, architecture and interfaces. The representations are then coded and tested for defects in function, logic and implementation. When the software is ready, it is implemented and maintained by support personnel. While each of the analysis, design, testing, implementation and maintenance phases need to be performed in all cases, there are a number of ways their interactions can be organized. In all the different models, design plays a central role in the models. Software design is thus a major phase in the software system development. In this research project, the design process of the SDLC will be considered, which includes requirement def inition or system analysis, system or requirement specifications, logical or system design, and detailed or program design and development. Even though pure software design consists of architectural and detailed design. Pure software design cannot proce ed without the requirement definition and specification stages. Architectural design deals with the general structure of a software system. It involves identifying and decomposing the software into smaller models or components, determining data structur es and specifying the relationship among the modules. According to a Software Engineering Institute report Budgen, , p6 , detailed design involves the "formulation of blueprints for the particular solution and with modeling the detailed interaction between its components. A Brief Survey of Software Design Approaches When a software designer looks at software problems that need to be solved, he will group those that have similar characteristics together. This grouping is called a problem domain. And for each software design methodology there is a group of problems for which it is well-suited called an application domain. The application domain is ill-defined, as such, it is impossible to make comparison amongst the software design methodologies, comparing each against another. However, some scheme of classifica tion will help in discussing particular methodologies. A few criteria can be used to classify methodologies such as the characteristics of the systems to be designed, such as the type of software representation and how formal it is. Other types of criteria are the characteristics of the software structure, whether it is hierarchical or level or it is functionally modular. Another criterion that can be used is the characteristic of the design process. However, the basis of the methodology is often the best guide to

classification. For example, a current classification scheme for real-time systems developed at the Software Engineering Institute Ford, , p53 is based on the three distinct views of a system. It can be summarized as follows: With the functional view, the system is considered to be a collection of components, each performing a specific function, and each function directly answering a part of the requirement. The design describes each functional component and the manner of its interaction with the other components. With the structural view, the system is considered to be a collection of components, each of a specific type, each independently buildable and testable, and able to be integrated into a working whole. Ideally, each structural component is also a functional component. With the behavioral view, the system is cons idered to be an active object exhibiting specific behaviors, containing internal state, changing state in response to inputs, and generating effects as a result of state changes. Dividing software design methodologies into classifications called approaches helps in the generalization, explanation and understanding of software design methodologies, and guide in the selection of the appropriate software design methodology to use. Details of software design approaches can vary greatly. Some consist of a set of guidelines, while others include strict rules and a set of coordinated diagrammatic representation. The approaches that have been proposed for software design are divers e. Many of these approaches are really full-fledged software design methods, in that they are composed of a set of techniques directed at and supporting a common, unifying rationale. The main design approaches Pressman, ; Peters, are: Different approaches have been taken to develop software solutions for different problems. However, in some problems, different approaches have been integrated or combined in a logical manner to derive a software solution. Thus, the different software design approaches are not necessarily mutually exclusive. For example, designers have used the leveling approach of top-down decomposition to break down a large complex system into smaller, more manageable modules and then use other approaches to design the software for each module. Depending on the criteria used, there are a number of ways to define the classification of software design approaches. The classification of software design approaches in this section is based on the basis of the approach, characteristics of the design process and the type of software constructs created to develop the solution. Level-Oriented Design In the level-oriented design approach, there are two general or broad strategies that can be used. The first strategy starts with a general definition of a solution to the problem then through a step-by-step process produce a detailed solution this is called Stepwise Refinement.

Chapter 3 : Free Online Course: Software Design Methods and Tools from Coursera | Class Central

DESIGN METHODOLOGIES - 2 A more methodical approach to software design is proposed by structured methods which are sets of notations and guidelines for software design.

One of the reasons for doing it this way is to discover non-obvious requirements that have a major impact on the high-level design of the program. Now we need to change the design of the program to include a module that identifies coins by their measurements. What usually happens in the real-world is that all of the top-down design is done on paper and the actual coding goes bottom-up. Then when the coding effort falsifies an assumption made on paper the high-level design is easier to change. Given a problem, there are certain traditional ways of solving it. Having windows on two adjacent walls in a room is a design pattern for houses that ensures adequate light at any time of day. The culture of computer programming has developed hundreds of its own design patterns: High-level patterns are baked into certain types of frameworks. The This is the first pattern you learned in programming class: One of the biggest mistakes made by programmers is to get too ambitious. But if you give the REPL the ability to maintain state between each command then you have a powerful base to build simple software that can do complex things. This is so you can easily adapt the program to work on terminals, GUIs, web interfaces and so-on. The Pipeline Input is transformed one stage at a time in a pipeline that runs continuously. A news aggregator that reads multiple feed standards like RSS and Atom , filters out dupes and previously read articles, categorizes and then distributes the articles into folders is a good candidate for the Pipeline pattern, especially if the user is expected to reconfigure or change the order of the stages. You write your program as a series of classes or modules that share the same model and have a loop at their core. The business logic goes inside the loop and treats each chunk of data atomically--that is, independent of any other chunk. Some language features that are useful for the Pipeline pattern are coroutines created with the "yield return" statement and immutable data types that are safe to stream. The Workflow This is similar to the Pipeline pattern but performs each stage through to completion before moving onto the next. A useful language feature to have is an easy way to serialize and deserialize the contents of the session to-and-from a file on disk, like to an XML file. The Batch Job The user prepares the steps and parameters for a task and submits it. The task runs asynchronously and the user receives a notification when the task is done. Yet even something as humble as printing a letter is also a batch job because the user will want to work on another document while your program is working on queuing it up. You need to encapsulate all of it into a self-contained data structure like "PrintJobCriteria" and "PrintJobResults" classes that share nothing with any other object and make sure your business logic has thread-safe access to any resources so it can run safely in the background. When returning results to the user you must make sure you avoid any thread entanglement issues. This is exactly the problem with the way file management is done in Windows: User Interface patterns A complete discussion of UI patterns would be beyond the scope of this tutorial, but there are two meta-patterns that are common between all of them: Modal means the user is expected to do one thing at a time, while modeless means the user can do anything at any time. A word processor is modeless, but a "Wizard" is modal. Regardless of what the underlying architectural pattern is, the UI needs to pick one of these patterns and stick to it. Almost all major desktop software is modeless and users tend to expect it. To support this kind of UI there are two mechanisms supported by most toolchains; event-driven and data binding. You attach a piece of code to a GUI control, and when the user does something to the control click it, focus it, type in it, mouse-over it, etc your code is invoked by the control. The call-back code is called a handler and is passed a reference to the control and some details about the event. Event-driven designs require a lot of code to examine the circumstances of the event and update the UI. The GUI is dumb, knows nothing about the data, needs to be spoon-fed the data, and passes the buck to your code whenever the user does something. Events are an important concept in business logic, overhead and models, too. The "PropertyChanged" event is very powerful when added to model classes, for example, because it fits so well with the next major UI mechanism below: Data Binding Instead of attaching code to a control you attach the model to the control and let a smart GUI read and manipulate the data directly. You can also attach

PropertyChanged and CollectionChanged events to the model so that changes to the underlying data are reflected in the GUI automatically, and manipulation on the model performed by the GUI can trigger business logic to run. Data binding is complemented with conventional event-driven style; like say your address-book controls are bound directly to the underlying Address object, but the "Save" button invokes a classic Event handler that actually persists the data. Data binding is extremely powerful and has been taken to new levels by technologies like Windows Presentation Foundation WPF. A perfectly usable viewer program can be built by doing nothing more than filling the model from the database and then passing the whole show over to the GUI. Or like MVC without the C. The style is part of a broader concept known as Reactive Programming. Designing models Remember that your data model should fit the solution, not the problem. Some of the best candidates for modelling first are actions and imaginary things, like transactions, requests, tasks and commands. CoinIdentifier has a public static method that takes input from the coin sensors in the machinery and spits out a value that identifies what kind of coin it is. The code which is the most likely to change frequently--like which coins to reject and how much to keep as a processing fee--stays in the main module, with constants like the processing fee read from a configuration file. Organizing business logic means grouping it by purpose, avoiding bundles of unrelated functions, and creating Interfaces that let us connect the modules while insulating them from changes in each other. The future maintainability of your code depends on it, but in large projects even the initial coding will succeed or fail on this principle. New patterns are being invented all the time, but these are the basic principles they all strive for: Overhead code needs to be as program-agnostic as possible Assume that every line of business logic and every property of the model will change and that the overhead code should expect it. You can do this by pushing that overhead code into its own classes and using Interfaces on models and business logic to define the very least of what the overhead needs to know about an object. You can also now find languages like Java, C and Visual Basic that support Generics, which extend the type-checking power of the compiler and by extension, the IDE. You can defer specifics with a design principle called Inversion of Control IoC. Event names that begin with Resolve often want the parent code to look for something it needs. An XML parser, for example, might use a ResolveNamespace event rather than force the programmer to pass all possible namespaces or a namespace resolving object at the beginning. This is when you identify a need--such as writing to different kinds of databases or output devices--and pass implementations of them into an object that uses them. Our Tabulator class from earlier, for example, could be passed an instance of the exact breed of CoinIdentifier in its constructor. We have one that works on US coins and another that works on Canadian, and a configuration file tells the main which one to create. The second method of DI is to call a static method from within the class that needs the resource. This is popular for logging frameworks: GetLogger in turn implements the Factory design pattern to pick and instantiate the appropriate kind of logger at runtime. With a tweak to a config file you can go from writing to text files to sending log events over the network or storing them in a database. To chose between the two I like to consider how critical the dependency is to the purpose of the module. A third method is to use a dependency injection framework like Ninject or Unity. Manual DI has a sweet spot of around 1-to-3 injections per object while frameworks start at about 4 or more.

Chapter 4 : How to design a computer program - Software Engineering Tips

A program design is also the plan of action that results from that process. Ideally, the plan is developed to the point that others can implement the program in the same way and consistently achieve its purpose.

Overview[edit] Software 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 begins by representing the totality of the thing that is to be built e. 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. Davis [3] suggests a set of principles for software design, which have been adapted and extended in the following list: The design process should not suffer from "tunnel vision. The design should be traceable to the analysis model. Because a single element of the design model can often be traced back to multiple requirements, it is necessary to have a means for tracking how requirements have been satisfied by the design model. The design should not reinvent the wheel. Systems are constructed using a set of design patterns, many of which have likely been encountered before. These patterns should always be chosen as an alternative to reinvention. Time is short and resources are limited; design time should be invested in representing truly new ideas by integrating patterns that already exist when applicable. The design should "minimize the intellectual distance" between the software and the problem as it exists in the real world. That is, the structure of the software design should, whenever possible, mimic the structure of the problem domain. The design should exhibit uniformity and integration. A design is uniform if it appears fully coherent. In order to achieve this outcome, rules of style and format should be defined for a design team before design work begins. A design is integrated if care is taken in defining interfaces between design components. The design should be structured to accommodate change. The design concepts discussed in the next section enable a design to achieve this principle. The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered. Well-designed software should never "bomb"; it should be designed to accommodate unusual circumstances, and if it must terminate processing, it should do so in a graceful manner. Design is not coding, coding is not design. Even when detailed procedural designs are created for program components, the level of abstraction of the design model is higher than the source code. The only design decisions made at the coding level should address the small implementation details that enable the procedural design to be coded. The design should be assessed for quality as it is being created, not after the fact. A variety of design concepts and design measures are available to assist the designer in assessing quality throughout the development process. The design should be reviewed to minimize conceptual semantic errors. There is sometimes a tendency to focus on minutiae

when the design is reviewed, missing the forest for the trees. A design team should ensure that major conceptual elements of the design omissions, ambiguity, inconsistency have been addressed before worrying about the syntax of the design model. Design Concepts[edit] The design concepts provide the software designer with a foundation from which more sophisticated methods can be applied. A set of fundamental design concepts has evolved. 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. 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 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. Default values for the parameters must be chosen so that they are a good choice for the majority of the users. Portability - The software should be usable across a number of different conditions and environments. Scalability - The software adapts well to increasing data or number of users. Modeling language[edit] A modeling language is any artificial language that can be used to express information, knowledge or systems in a structure that is defined by a consistent set of rules. These rules are used for interpretation of the components within the structure. A modeling language can be graphical or textual. Examples of graphical modeling languages for software design are: Flowchart is a schematic representation of an algorithm or step-wise process. Jackson Structured Programming JSP is a method for structured programming based on correspondences between data stream structure and program structure. Unified Modeling Language UML is a general modeling language to describe software both structurally and behaviorally. It has a graphical notation and allows for extension with a Profile UML. Alloy specification

language is a general purpose specification language for expressing complex structural constraints and behavior in a software system. It provides a concise language base on first-order relational logic.

Chapter 5 : Software Design Methodology

A few mature and popular methods are currently being used to specify and design real-time embedded systems software, and these methods are the basis for a large number of tools automating the process. Unfortunately, some of the tools support only parts of a method, while others support a mixture of.

About Program Design and Evaluation: Here are some short answers to common questions that I field. What is a program? When organizations set out to make the world a better place, they develop a plan of action. That plan is a program. Historically, nonprofit organizations have implemented programs underwritten by foundations and government agencies. Examples include museums, schools, and social services. Governmental organizations also implement programs directly. Education and health are two areas in which government implements programs at the local, state, and federal levels. Today, a growing number of for-profit corporations are implementing programs to advance their social missions. Programs are tremendously varied. They may work with individuals like an afterschool program or organizations like foundations that support afterschool programs. They may focus on service delivery like a school district or policy that gives rise to service delivery like political advocacy groups. What they hold in common is thisâ€”they are concrete plans of action implemented by organizations with the intention to make the world a better place. What is program design? Program design is both a verb and a noun. It is the process that organizations use to develop a program. Ideally, the process is collaborative, iterative, and tentativeâ€”stakeholders work together to repeat, review, and refine a program until they believe it will consistently achieve its purpose. A program design is also the plan of action that results from that process. Ideally, the plan is developed to the point that others can implement the program in the same way and consistently achieve its purpose. The more energy, creativity, and hard work that goes into program design, the greater the chances that a program will succeed. What is program evaluation? Program evaluation is an organized effort to understand how effective a program is and how it can be made more effective. It can be undertaken in a formative manner that supports the program design processes. Or it can be undertaken in a summative manner that measures the effectiveness of a design as implemented in a given context. Typically, program evaluation shifts from formative to summative, and from informal to rigorous, as the program design process advances.

Chapter 6 : Software design - Wikipedia

A software design methodology can be structured as comprising of the software design process component and the software design representation or diagrammatic component. The process component is based on the basic principles established in the methodolog y while the representation component is the "blueprint" from which the code for the software.

Coding standards or coding conventions Sustainable pace i. The core practices are derived from generally accepted best practices, and are taken to extremes: Interaction between developers and customers is good. Therefore, an XP team is supposed to have a customer on site, who specifies and prioritizes work for the team, and who can answer questions as soon as they arise. In practice, this role is sometimes fulfilled by a customer proxy. If learning is good, take it to extremes: Reduce the length of development and feedback cycles. Simple code is more likely to work. Therefore, extreme programmers only write code to meet actual needs at the present time in a project, and go to some lengths to reduce complexity and duplication in their code. If simple code is good, re-write code when it becomes complex. Code reviews are good. Therefore XP programmers work in pairs, sharing one screen and keyboard which also improves communication so that all code is reviewed as it is written. Testing code is good. Therefore, in XP, tests are written before the code is written. The code is considered complete when it passes the tests but then it needs refactoring to remove complexity. The system is periodically, or immediately tested using all pre-existing automated tests to assure that it works. It used to be thought that Extreme Programming could only work in small teams of fewer than 12 persons. However, XP has been used successfully on teams of over a hundred developers. Peter describes FDD as having just enough process to ensure scalability and repeatability while encouraging creativity and innovation. More specifically, Feature Driven Development asserts that: A system for building systems is necessary in order to scale to larger projects. A simple, but well-define process will work best. Process steps should be logical and their worth immediately obvious to each team member. Good processes move to the background so team members can focus on results. Short, iterative, feature-driven life cycles are best. FDD proceeds to address the items above with this simple process numbers in brackets indicate the project time spent: Develop an overall model 10 percent initial, 4 percent ongoing 2. Build a features list 4 percent initial, 1 percent ongoing 3. Plan by feature 2 percent initial, 2 percent ongoing 4. Design by feature 5. So the Joint Application Development JAD methodology aims to involve the client in the design and development of an application. This is accomplished through a series of collaborative workshops called JAD sessions. JAD focuses on the business problem rather than technical details. It is most applicable to the development of business systems, but it can be used successfully for systems software. Its success depends on effective leadership of the JAD sessions; on participation by key end-users, executives, and developers; and on achieving group synergy during JAD sessions. In contrast to the Waterfall approach, JAD is thought to lead to shorter development times and greater client satisfaction, both of which stem from the constant involvement of the client throughout the development process. On the other hand, with the traditional approach to systems development, the developer investigates the system requirements and develops an application, with client input consisting of a series of interviews. Rapid application development RAD , a variation on JAD, attempts to create an application more quickly through strategies that include fewer formal methodologies and reusing software components. This methodology embodies the notion of dynamic stability which can be thought of as similar to how Scrum embraces controlled chaos. Bob Charette, the originator, writes that the measurable goal of LD is to build software with one-third the human effort, one-third the development hours and one-third the investment as compared to what SEI Software Engineering Institute CMM Level 3 organization would achieve. There are 12 principles of Lean Development: Satisfying the customer is the highest priority. Always provide the best value for the money. Success depends on active customer participation. Every LD project is a team effort. Domain, not point, solutions. An 80 percent solution today instead of percent solution tomorrow. Product growth is feature growth, not size growth. Never push LD beyond its limits. Rapid-Development Languages RDLs produce their savings by reducing the amount of construction needed to build a product.

Although the savings are realized during construction, the ability to shorten the construction cycle has projectwide implications: Because RDLs often lack first-rate performance, constrain flexibility, and are limited to specific kinds of problems, they are usually better suited to the development of in-house business software and limited-distribution custom software than systems software. RAD rapid application development proposes that products can be developed faster and of higher quality by: Using workshops or focus groups to gather requirements. Prototyping and user testing of designs. Following a schedule that defers design improvements to the next product version. Keeping review meetings and other team communication informal. There are commercial products that include requirements gathering tools, prototyping tools, software development environments such as those for the Java platform, groupware for communication among development members, and testing tools. RAD usually embraces object-oriented programming methodology, which inherently fosters software re-use. This process recognizes that the traditional waterfall approach can be inefficient because it idles key team members for extended periods of time. Many feel that the waterfall approach also introduces a lot of risk because it defers testing and integration until the end of the project lifecycle. Problems found at this stage are very expense to fix. By contrast, RUP represents an iterative approach that is superior for a number of reasons: It lets you take into account changing requirements which despite the best efforts of all project managers are still a reality on just about every project. Risks are usually discovered or addressed during integration. With the iterative approach, you can mitigate risks earlier. Iterative development provides management with a means of making tactical changes to the product. It allows you to release a product early with reduced functionality to counter a move by a competitor, or to adopt another vendor for a given technology. Iteration facilitates reuse; it is easier to identify common parts as they are partially designed or implemented than to recognize them during planning. When you can correct errors over several iterations, the result is a more robust architecture. Performance bottlenecks are discovered at a time when they can still be addressed, instead of creating panic on the eve of delivery. Developers can learn along the way, and their various abilities and specialties are more fully employed during the entire lifecycle. Testers start testing early, technical writers begin writing early, and so on. The development process itself can be improved and refined along the way. The assessment at the end of iteration not only looks at the status of the project from a product or schedule perspective, but also analyzes what should be changed in the organization and in the process to make it perform better in the next iteration. Scrum Methodology Scrum is an agile method for project management developed by Ken Schwaber. Its goal is to dramatically improve productivity in teams previously paralyzed by heavier, process-laden methodologies. Scrum is characterized by: A living backlog of prioritized work to be done. Completion of a largely fixed set of backlog items in a series of short iterations or sprints. A brief daily meeting called a scrum , at which progress is explained, upcoming work is described, and obstacles are raised. A brief planning session in which the backlog items for the sprint will be defined. A brief heartbeat retrospective, at which all team members reflect about the past sprint. Scrum is facilitated by a scrum master, whose primary job is to remove impediments to the ability of the team to deliver the sprint goal. The scrum master is not the leader of the team as they are self-organizing but acts as a productivity buffer between the team and any destabilizing influences. Scrum enables the creation of self-organizing teams by encouraging verbal communication across all team members and across all disciplines that are involved in the project. Spiral Methodology The Spiral Lifecycle Model is a sophisticated lifecycle model that focuses on early identification and reduction of project risks. A spiral project starts on a small scale, explores risks, makes a plan to handle the risks, and then decides whether to take the next step of the project - to do the next iteration of the spiral. It derives its rapiddevelopment benefit not from an increase in project speed, but from continuously reducing the projects risk level - which has an effect on the time required to deliver it. Success at using the Spiral Lifecycle Model depends on conscientious, attentive, and knowledgeable management. It can be used on most kinds of projects, and its risk-reduction focus is always beneficial. The spiral methodology extends the waterfall model by introducing prototyping. It is generally chosen over the waterfall approach for large, expensive, and complicated projects. At a high-level, the steps in the spiral model are as follows: The new system requirements are defined in as much detail as possible. This usually involves interviewing a number of users representing all the external or internal users and other

aspects of the existing system. A preliminary design is created for the new system. A first prototype of the new system is constructed from the preliminary design. This is usually a scaled-down system, and represents an approximation of the characteristics of the final product.

Chapter 7 : Software Development Methodologies

Master of Design Methods Overview The Master of Design Methods (MDM) program is for exceptional professionals of any discipline with proven innovation and design experience. Graduates further accelerate their careers by building lasting skills and knowledge applicable to a wide range of challenges within large and small enterprises.

Determine if the existing facility is satisfactory or obsolete as a resource. If a floor plan exists, do a square foot take-off of the areas for various functions. Determine the building efficiency the ratio of existing net-to-gross area. This ratio is useful in establishing the building efficiency target for the new facility. If the client is a repeat builder school districts, public library, public office building, etc. Use the existing square footages for comparison when you propose future amounts of space. People can relate to what they already have. See illustration above in Step 5, Determine quantitative requirements. A familiar example of a programmatic strategy is the relationship or "bubble" diagram. These diagrams indicate what functions should be near each other in order for the project to function smoothly. Relationship diagrams can also indicate the desired circulation connections between spaces, what spaces require security or audio privacy, or other aspects of special relationships. Other types of strategies recur in programs for many different types of projects. Some examples of common categories of programmatic strategies include: What function components are grouped together and which are segregated? For example, in some offices the copying function is centralized, while in others there are copiers for each department. What types of changes are expected for various functions? Do facilities need to change over a period of a few hours? Or is an addition what is really needed? What goods, services, and people move through the project? What is needed at each step of the way to accommodate that flow? What are the most important functions of the project? What could be added later? Are there ongoing existing operations that must be maintained? Who is allowed where? What security levels are there? Ideally, each of the goals and objectives identified in Step 2 will have some sort of strategy for addressing that goal. Otherwise, either the goal is not very important, or more discussion is required to address how to achieve that goal or objective. Costs are affected by inflation through time. Affordable area is determined by available budgets. In this step, one must reconcile the available budget with the amount of improvements desired within the project time frame. First, a list of spaces is developed to accommodate all of the activities desired see Exhibit A. The space criteria researched in Step 3 are the basis of this list of space requirements. The space requirements are listed as net assignable square feet NASF , referring to the space assigned to an activity, not including circulation to that space. A percentage for "tare" space is added to the total NASF. Tare space is the area needed for circulation, walls, mechanical, electrical and telephone equipment, wall thickness, and public toilets. The building efficiency for a building type was researched in Step 1 and possibly Step 3. See Exhibit A for an example of space requirements. The building efficiency of an existing space used by a client can inform the selection of the net-to-gross ratio. The example below of an office suite within an office building illustrates the areas of net assignable square feet and tare area. Notice that some space within an office is considered circulation, even though it is not delineated with walls. We call this circulation, "phantom corridor. Additional support space or tare area such as mechanical rooms and public toilets would not be included in the calculation for this project type. In drafting the total project cost, the programmer uses the cost per square foot amount researched in Step 1. Factors for inflation should be included, based upon the project schedule. Costs should be projected to the date of the mid-point of construction because bidders calculate estimates on the assumption that costs could change from the time of the bid date. The intention is to help the owner prepare for all the project costs, not just those costs assigned to construction. If the bottom line for the project costs is more than the budget, three things can happen: This reconciliation of the desired space and the available budget is critical to defining a realistic scope of work. All of the pertinent information included above can be documented for the owner, committee members, and the design team as well. The decision-makers should sign-off on the scope of work as described in the program. Once a program is completed and approved by the client, the information must be integrated into the design process. Some clients want the programmer to stay involved after the programming phase to insure that the requirements defined in the program are realized in the design

work. Emerging Issues Some of the emerging issues in the discipline of architectural programming include: Development of standards and guidelines for owners that build similar facilities frequently. Formalizing computerizing building facility requirements for Web-based consumptionâ€"for example, the National Park Service has developed Facility Planning Model Web-based software to assist park superintendents and other staff in the development of space and cost predictions for legislative requests. The intention is to make budget requests more realistic and more comprehensive. Facility programming to make early predictions to aid in early capital budgeting Client-owners are increasingly requiring verification that the design complies with the program. New technologies are generating a need for types of space which have no precedents. Basic research on these technologies is required to determine standards and guidelines. As more clients require measures for building energy and resource conservation standards LEED, Green Globes, etc , the programming process needs to reflect these requirements in goals, costs, scheduling, and process. The supply of facility programmers is smaller than the demand. More professionals need to consider this sub-discipline as a career path. Relevant Codes and Standards A very important part of programming is identifying relevant codes and standards that apply to the project see Steps 1 and 3 above. Codes, covenants, deed restrictions, zoning requirements, licensing requirements, and other legal obligations can have significant influence on costs and therefore, affordable GSF. These factors must be identified prior to design. Many governments and institutions have developed standards and guidelines for space allocations. For example, the General Services Administration GSA , military, and higher education institutions all have standards and guidelines. These standards must be adhered to in programming projects for these clients. The standards are also useful as guidelines for agencies that have not developed their own standards. Some standards are mandated by statutes in some jurisdictions for licensing, accreditation, or equity purposes. Schools, hospitals, correctional facilities, and other licensed or accredited institutions may be required to meet these standards prior to opening their doors. Some building codes identify the number of square feet allocated per person for certain types of occupancy. However, while these ratios may determine the legal occupancy numbers for the facility, exiting requirements, fire separations, etc. It may be necessary to accommodate specific activities adequately with more space.

Chapter 8 : Basic Guide to Nonprofit Program Design and Marketing

Software design is a process to conceptualize the software requirements into software implementation. Software design takes the user requirements as challenges and tries to find optimum solution. While the software is being conceptualized, a plan is chalked out to find the best possible design for.

The word "formal" means the use of a formal language, so that the program logic can be machine checked. Our compilers already tell us if we make a syntax error, or a type error, and they tell us what and where the error is. Formal methods take the next step, telling us if we make a logic error, and they tell us what and where the error is. And they tell us this as we make the error, not after the program is finished. It is good to get any program correct while writing it, rather than waiting for bug reports from users. It is absolutely essential for programs that lives will depend on. The course begins by introducing or reviewing the basic logic that will be used as an aid to programming. Then we look at formal specifications, and how they are refined to become programs. At each refinement step, there is a small theorem to be proven that the step is correct , so that at the end, the program is correct. Most of the course uses just those programming constructs that are common to most programming languages assignment statement, if statement, array. We also look at parallel and interacting processes, at probabilistic programming, and functional programming. Along the way, we formally define the language features we use both execution control and data structures. The emphasis is on program development to meet specifications, and on program modifications that preserve correctness, rather than on verification after a program is finished. The four best motivations for this course are written here , here , here , and here. Get the course textbook. Here are pages ,.. Here are some study questions. And here is some tutorial material for Chapter 1 , Chapter 3 , and Chapter 4. Here are the solutions to the exercises. This course comes in two versions: You must be registered at a university that offers this course. There is a start date and a finish date for the course. There are tests during the term and an exam at the end. The times and places of the tests and exam are set by the university. You may start the course whenever you like, proceed at your own pace, and finish whenever you like. There are no tests or exams. Whether you are taking the course for-credit or not-for-credit, it is very valuable for your understanding of the course material to do written exercises assignments as you go through the course. With each lecture segment, there is a suggested small exercise or two from the textbook, and there are hundreds more exercises in the textbook for you to choose from. You may work on them by yourself or in a group. In the for-credit version, there is classroom time and an online chat room for students to discuss the exercises and course material with the course instructor and with each other. In the not-for-credit version, you may email questions to me. If you choose to typeset your solutions in LaTeX, here is a helpful set of macros. When you have done an exercise, compare your answer to the solution online. If you just look up the solution without trying the exercise first, you will not get the full benefit of the exercise. Solutions to exercises will not be graded.

Chapter 9 : Software Design Methods | CSIAC

Firmly tie your methods to the proposed program's objectives and needs statement. Link them to the resources you are requesting in the proposal budget. Explain why you chose these methods by including research, expert opinion, and your experience.

Resources and Activities Organized to Provide Related Services Basically, a nonprofit program is a highly integrated set of resources and activities geared to provide a service or closely related set of services to clients. The typical nonprofit organizational structure is built around programs. Two other major aspects of the nonprofit structure are its governance the board and, for some, the chief executive, too and its central administration. The board oversees the entire nonprofit organization. Programs are not the same as activities!!! Many nonprofits have activities, not programs. The following article describes the difference: For more information now about nonprofit organizations, their structures and nature, see Overview of Nonprofit Organizations Program "System": Inputs, Processes, Outputs and Outcomes Programs, like other organizations, can seem a highly confusing, amorphous mess that is very hard to comprehend. It can be hard to keep perspective. However, like the overall organization itself, a program is a system with inputs, processes, outputs tangibles and outcomes impacts on clients -- with ongoing feedback among these parts. This systems perspective helps keep clarity about programs and will help a great deal during program planning. Program inputs are the various resources needed to run the program, e. The processes are how program services are delivered, e. The outputs are the units of service, e. Outcomes are the impacts on the clients who are receiving the services, e. The outcomes are the "compass" for the program and help it keep its direction. This is why funders are increasingly requesting outcomes-based evaluations from nonprofits. For more information now about outcomes-based evaluation, see Program Evaluation. During strategic planning, planners work from the mission to identify several overall, major or strategic goals that must be reached and that, in total, work toward the mission. Each program is associated with achieving one or more strategic goals and, therefore, should contribute directly toward the mission as well. If an idea for a program comes up at some time other than during the strategic planning process, nonprofit board members must carefully ask themselves if the program is really appropriate to the mission of the organization. Goals associated with services to clients often become program s and strategies to reach those goals often become methods of delivering services in the programs. Typically, at a point right after the strategic planning process has identified strategic goals and issues, a team of planners can draft a framework for how strategic goals can be met. This framework is often the roadmap for a new program. For more information now about strategic planning, see Strategic Planning Nonprofits must strive to keep down overhead costs, which are often interpreted as the costs of central administration. Therefore, avoid developing programs to fix recurring administrative problems in the workplace. Involve Board Members in Program Planning A major responsibility of board members is to set the strategic direction for their nonprofit. Therefore, board members should be highly involved in the strategic and program planning processes in the nonprofit. However, staff members might be strongly involved in determining how services will actually be delivered in the program. For more information now about the roles and responsibilities of board. Conduct Program Planning as a Team The chief executive, key planners on the board, relevant middle managers and representatives from major client groups should all be involved in program planning. Therefore, involve clients as much as possible in initial ideas for a program. Discuss with them your perceptions of their unmet needs. Try verify if these needs actually exist and how they would like their needs to be met. You might have representatives from client groups review the final draft of your program plan. Note that this involvement of clients is a critical aspect of the marketing process, specifically marketing research. The organization remains the only real "expert" on their own planning. Outside consultants and facilitators can be brought in, but each planning decision is ultimately up to the organization members. The "perfect" program plan will meet the nature and needs of the organization and continue to be updated as organization members learn more about meeting the needs of their clients. You can change your plans -- just know why and be able to explain e. This planning effort is almost always more than nonprofit personnel want to undertake, but is almost always less

than they fear. Outcomes, Goals, Strategies and Objectives Program Outcomes, Goals and Strategies Follow Directly from Strategic Planning If your strategic planning was done thoroughly then it should be relatively easy to determine program outcomes, goals and strategies. The strategic planning process determines the mission or purpose of the organization in terms of uniquely accomplishing certain outcomes for specific groups of clients. The process also determines the goals needed to work towards the mission and the general methods or strategies to reach the goals. As much as possible, goals are specified in terms to meet specific needs among specific groups of clients. The overall goals of the organization very much determine whom you want to serve, that is, who your target markets will be. For example, strategic goals might be to expand the number of clients you have now, get new clients, get more revenue from current clients, etc. You may want to develop new services in a current or new market, or expand current services in a current or new market. These examples of strategic goals greatly determine who your target markets will be. You might consider goals as measurable accomplishments and objectives as smaller, measurable milestones along the way to the goals. Consider strategies as methods to reach the goals or objectives. Outcomes are benefits to clients from participation in the program. Example Outcome 1 -- Drop-outs from Minneapolis high schools obtain high school diplomas or equivalent levels of certification Example Outcome 2 -- Within three months after getting certification, participants obtain at least half-time employment or enroll in an accredited program to further their education Program Goals Programs goals should follow directly from, or be the same as, strategic service goals intended to meet specific needs of specific client groups. Note that there are also strategic goals other than for meeting needs of clients, for example, getting a facility. Goals should specify the results from program services and be in terms that are "SMARTER" an acronym , that is, specific, measurable, acceptable to those working to achieve the goals, realistic, timely, extending the capabilities of those working to achieve the goals and rewarding for them, as well. Example Program Goal 1: Support at least drop-outs from Minneapolis high schools to obtain diplomas or equivalent levels of certification Program Strategies Program strategies or methods to reach goals should follow directly from strategies intended to achieve each strategic goal, for example: Example Program Strategy 1. Nonprofit services must be marketed, including clarifying which client groups the nonprofit is going to serve these are target markets , verifying their needs a basic form of market research , analyzing competitors nonprofits do have competitors and potential collaborators, determining the best fee for services, determining how to produce and distribute the services, and how to promote advertise, manage public image and sell the services, as well. Draft Basic Description of Each of Your Services Typically, a service is a closely related set of activities that accomplishes a specific benefit for clients. Exactly what determines a service in an organization is highly unique to the organization itself. A program can have several services. For example, from the above example outcome 1, services might include: Example Service for Outcome 1 -- High-school training services Example Service for Outcome 1 -- Transportation services Example Service for Outcome 1 -- Child-care services By now, you might have a strong, clear sense of what each of your services are. At this point, you might draft for yourself a written description of each of your services. The description should include: Be careful to describe the services in terms of benefits to clients, not to you. For example, address pricing, convenience, location, quality, service, atmosphere, etc. Target Markets and Customer Profiles This paragraph is repeated from above: The overall goals of the organization very much determine whom you want to serve. In addition to helping focus the results and evaluation of your services, understanding your target markets helps you to focus on where to promote your services, including advertising, conducting public relations campaigns and selling your services. However, it is very useful to determine several additional target markets. These additional markets are often where you should focus promotions and mean additional sources of assistance and revenue. For example, a target market that follows from the above examples, might be: Dropouts from Minneapolis high schools Target Market 2: Counselors in Minneapolis high schools Target Market 3: Parents of drop-outs from Minneapolis schools Target Market 4: Job placement services, seeking to help people find jobs Target Market 5: Local businesses looking for employees The more you know about your clients, the better you might be at serving them. At this point, write down a customer profile, or description of the groups of clients or markets who will use your services. Consider, for example, their major needs, how they prefer to have their needs met,

where they are and where they prefer to have their needs met and demographics information their age ranges, family arrangement, education levels, income levels, typical occupations, major interested, etc. By now, you should have clear idea of the major needs met by each of your services for their primary or 1 target market. What needs might your services meet among other markets as well? For example, in the above examples: Needs Met for Target Market 1: High school graduation, eligibility for job and further education Needs Met for Target Market 2: Place to refer high school drop-outs so they can continue their education Needs Met for Target Market 3: Place to refer their children for continued training, transportation and child care Needs Met for Target Market 4: Place to get clients to find jobs for Needs Met for Target Market 5: Place to get job candidates Note that identifying the needs of target markets is a major aspect of the marketing process, specifically marketing research. Nonprofits exist to serve their communities. One would think that in this spirit of service, all nonprofits should collaborate for "the common good". However, nonprofits do compete for the attention, participation and money of their clients -- and in many cases, compete for the same items from funders. Consider the following questions: Who are your competitors? What client needs are you competing to meet? What are the strengths and weaknesses of each of their products and services? How do their prices compare to yours?