System Design

teodoraristic

November 2021

0.1 An architectural design example

Designing the house:

- Starts after getting an agreement with the customer regarding the number of rooms, floors, size and location of the house
- The goal is to set a floor plan that defines the arrangement of rooms/ walls, position of windows and doors, heating, water pipes, drain
- The architect must consider several standards (standard (kitchen) element/cabinet dimensions, standard bed sizes. . .)
- The architect must not limit the actual furnishing

Architectural design starting point (SRS):

- The house must have two bedrooms, study room, kitchen, living room... (functional requirements)
- The distance walked by the residents shall be as small as possible (nonfunctional requirement)
- Maximum use of daylight (non-functional requirement)
- Best use of space (non-functional requirement).

Similarities with software architecture design:

- The whole system gets divided into less complex components (house→ rooms == system → components (subsystems)
- Connections between components are important (doors, corridors == interfaces)
- Non-functional requirements (room size == performance)
- Functional requirements (required rooms == use cases)
- System design influences component design and implementation (kitchen plan == component design)
- Difficulty (and price) of later changes (moving the walls == changing interfaces)
- Component design is left for further design stages (furnishing plan == component design)

0.2 System design

System design is an activity of translating an SRS into a system design model. **The System Design Model divides the system into subsystems (components), describe their behavior, their relationship and interfaces.** It includes all the decisions that influence the structure of the whole system, also known as **Software Architecture**. System design isn't algorithmic.

A **stable architecture** assures reliability of a system and easy maintenance. It enables adding functionalities with minimal architecture changes.

System design includes:

1. Definition of design goals
2. Decomposition of a system into subsystems
3. Selection of (already developed) reusable components
4. Relation between SW and HW
5. Selection of infrastructure for persistent data management
6. Definition of the access control policy
7. Selection of a global control flow mechanism
8. Handling borderline cases

1 Design goals

Definition of design goals is the first step of system design. Most design goals follow the nonfunctional requirements or application domain. Other design goals must be coordinated with the customer. In general, all design goals can be selected from a predefined list of general software product quality wishes that could be grouped into five categories: **performance, dependability, cost, maintenance, user criteria.**

1.1 Design goals - performance 3

1. **response time** - time between request and response
2. **throughput** - number of tasks executed in a time unit
3. **memory** - how much memory is required for normal operation?

1.2 Design goals - dependability 6

1. **robustness** - ability to survive irregular user request (input flow)
2. **reliability** - difference between specified and observed behavior
3. **availability** - percentage of time when system can be used for user tasks
4. **fault tolerance** - ability to operate under wrong conditions
5. **security** - ability to resist malicious attacks
6. **safety** - ability not to endanger human lives, even in case of errors and defects

1.3 Design goals - cost 5

1. **development cost** - cost of development of the initial system
2. **deployment cost** - cost of installation and user education
3. **upgrade cost** - cost of data translation and system extensions
4. **maintenance cost** - cost of error correction and system extensions
5. **administration cost** - cost of system administration

1.4 Design goals - maintenance

1. **extensibility** - how difficult it is to implement additional functionality
2. **modifiability** - how difficult it is to change existing functionality
3. **adaptability** - how difficult it is to transfer the system to other user domains
4. **portability** - how difficult it is to transfer the system to other platforms
5. **readability** - how difficult it is to understand the system by reading the source code
6. **traceability of requirements** - how difficult is to link the code with specific requirement

1.5 Design goals - user criteria

1. **utility** - how well the system supports user's work
2. **usability** - how difficult it is to use the system by users

1.6 Design goals contradictions

At the same time, only a part of all criteria can be satisfied (e.g. it's not realistic to expect a system that's secure and inexpensive). The developer typically defines priorities for the design goals and uses them to ease the decisions during system development.

1.7 Opposing design goals

1. **memory vs throughput** - Throughput and response time can be increased at the expense of memory usage
2. **delivery time vs functionality** - Faster delivery times can be ensured by limiting system functionality
3. **delivery time vs quality** - In the absence of time, the decision may be made to hand over a product with known defects
4. **delivery time vs number of developers** - Shortening delivery times by increasing the number of developers is only reasonable in the initial stages of development, otherwise, by increasing the number of developers, the development time can even be extended (due to need for training, extensive communication...)

2 Software system decomposition

2.1 Divide and conquer

Dealing with large systems is much more difficult than dealing with a series of small components - subsystems. Smaller components are easier to understand than large ones. Each subsystem can be taken care of by other people. There's possibility of specialization of individual software engineers. Components can later be replaced without major repairs to the rest of the system.

2.2 Identification of subsystems

Identifying subsystems requires system designer to be inventive (finding objects in the specification stage of a requirements analysis). Each additional request may combine several subsystems into one subsystem, divide complex subsystems into several subsystems, adds subsystems that provide additional functionality.

Possible heuristic approach:

1. objects from the same use case should be part of the same subsystem
2. a separate subsystem should be included to transfer data between subsystems
3. minimize the number of connections between subsystems
4. keep all objects in the subsystem functionally connected

2.3 The concept of subsystems

Subsystems provide services to other subsystems. The **service** is a group of related operations. When designing a system (architecture), it's necessary to define subsystems in terms of the services that subsystems provide. The subsystem operations are available to other systems from the subsystem interface, also called the application programmer interface (API).

2.4 Service planning

When designing the system, the services of each subsystems must be defined: list operations, determine their parameters, determine their higher-level behavior. Only later, during component design the API is updated with type of parameters and types of data returned by each operation.

2.5 Decomposition of software systems

System→subsystem→packages→classes→methods

The distributed system is divided into clients and servers. The system is subdivided into subsystems. The subsystem may consist of one or more packages. The package consists of classes. The class contains several methods.

2.6 Layers and partitions

Layers separate the subsystems vertically and **partitions** separate subsystems of the same layer horizontally. Each subsystem (layer of partition) is responsible for different services. The subsystems are weakly interconnected and often operate independently.

The layers form a hierarchical structure. The layer provides services to the upper layers and depends only on the lower layers:

- **closed architecture** - the layer uses only the services of the first lower layer
- **open architecture** - the layer can use the services of all the lower layers

A layer can be replaced without affecting other layers.

Layers in complex systems provide insight into the system at different levels of abstraction. It's important to have a separate layer dedicated to the user interface (UI). The layer under UI provides functionality defined by use cases. The lowest layer provides general services (eg. network communications, database access...)

2.7 Subsystem coupling

Coupling refers to the relationship between two subsystems. If the two subsystems are strongly interconnected, changes to one subsystem will have an impact on the operation of the other subsystem. It's desirable that the subsystems are connected as weakly as possible because this reduces the impact of errors and changes of one subsystem on the performance of another subsystem.

There are several types of coupling:

1. **Content coupling** - when one component secretly changes the internal data of another component. To reduce content coupling, it's suggested to encapsulate all instance variables: defining variables as private and providing get and set methods.
2. **Common coupling** - when global variables are used. All components that use a global variable become interconnected. They can be acceptable if the global variables represent the system default values. For global access to an object, a singleton design pattern may be used.
3. **Control coupling** - if one procedure calls another procedure, specifying a command that explicitly defines the operation of the other procedure. In case of changes, both procedures need to be changed. Control connectivity can often be avoided by using polymorphic operations (the same operation is performed differently for different input types/classes).
4. **Stamp coupling** - when one of the classes is used as an argument type of a method in the other class. Because one class uses the other, system changes are more difficult: changes are required for both classes and reusing a class requires reusing both classes. There are two ways to eliminate this kind of coupling: use of interface for an argument type and use of simple variables (primitive types).

5. **Data coupling** - when argument types of methods are primitive or simple library classes. Calling a method from another class requires multiple data to be provided. The more arguments there are, the greater the coupling. Coupling can be reduced by not using unnecessary arguments. Data coupling is related to stamp coupling - reducing one typically increases the other.

6. **Routine call coupling** - when one routine (or another method in the OO system) calls another. Routines are coupled because one routine depends on the behavior of the other routine. Routine connectivity is always present, in every system. If a sequence of two or more methods (routines) is used repeatedly, routine coupling can be reduced by constructing a routine that encapsulates the sequence.

???

7. **Type use coupling** - when a module uses the data type specified in another module. Using the class of the second module as the data type for the declaration of variables. In the case of changing the definition of a data type, this may also require a change in the procedures that use this type. For a OO system, the most general class (or interface) containing the required operations should be used.

8. **Inclusion or import coupling** - when one component includes a package (import in Java) or another component (include in C++). A component is subject to everything that is contained in the included component. Changes or additions to the included component can lead to conflicts (conflicts in variable names, functions, definitions ...).

9. **External coupling** - where the module depends on the operating system, shared libraries or hardware. The number of places where such dependencies occur should be kept to a minimum. External coupling can be reduced by using a facade design pattern.

2.8 Cohesion of subsystems

Cohesion refers tot he internal relations inside a subsystem. Cohesion is high if the subsystem combines the interdependent or linked components and doesn't contain other, unrelated components. Greater cohesion of the subsystem is desirable, because such subsystems are easier to understand and modify.

There are several types of cohesion: 7

1. **Functional cohesion** - it's obtained when all the code that is intended to produce a specific result is combined, while the rest of the code isn't included. When the module is intended to perform a single operation. Advantages for the system: easier to understand, increased reusability, easier component replacement.

2. **Layer cohesion** - the layer includes all that is needed to provide or access a related service (and nothing else is included).

3. **Communication cohesion** - all modules that access or use certain data are grouped together (eg. into one class), nothing else is included. The class has good communication cohesion if it includes all system operations associated with storing and managing its data and if it doesn't perform operations unrelated to the management of its data. The main advantage is that when data related changes are required, all relevant code is located in one place.

4. **Sequential cohesion** - grouping procedures that follow one after the other - one procedure provides input for the other procedure, the rest isn't included. The prerequisite for ensuring sequential cohesion is to satisfy all of the previously mentioned cohesion types.

5. **Procedural cohesion** - the procedures that are used one after the other are combined. Even if one procedure is not related by the inputs/outputs (one procedure doesn't provide the input to the other). Procedural cohesion is weaker than sequential.

6. **Temporal cohesion** - operations that are executed at the same stage of software execution are combined, nothing else is included. Temporal cohesion is weaker than the procedural one.

7. **Utility cohesion** - operations that can't logically be integrated into other cohesive units are grouped. A utility is a procedure or class that's more widely used across multiple subsystems and is designed to be reusable (eg. class java.lang.Math)

2.9 Dependency models

Dependency models help to analyze software architectures in terms of identifying problematic dependencies between modules. There are two ways to look at module dependencies: **dependency graph** and **dependency matrix**.

3 Relation between hardware and software

Many systems run on multiple computers that communicate with each other over an intranet or the Internet. Distribution of system components across computers (nodes) and communication infrastructure plan is required. Often, additional subsystems are required to divide subsystems by nodes: communication subsystems designed to transmit data and ensure transaction concurrency and control systems designed to ensure reliability.

By arranging subsystems across nodes, functionality and processing power can be distributed to be available where we need it or to be used where available. The challenges of storing, transferring duplication and synchronizing data.

4 Persistent data management

Persistent data isn't lost when ending or closing the software. Where and how the data is stored affects the decomposition of the system. In some cases (eg. shared repository architecture example) the entire subsystem is intended for this purpose. The first step in data storage is planning to identify persistent data, followed by a decision on how to store it: what data should be persistent? How to access persistent data?

Persistent data management is often bottleneck for system performance: access to data must be fast and reliable. It often requires the selection of a database management system or additional subsystems dedicated to persistent data management.

Options:

1. **Direct use of a file system (flat files)** - access to data at a relatively low level. The application itself has to solve the problems of simultaneous access to data, data loss due to downtime
2. **Relational databases** - The level of abstraction already ensured in accessing the data. May be a performance bottleneck
3. **Object-oriented databases** - Higher level of abstraction. They store data in the form of objects and links. Typically slower performance than relational bases.

Persistent data storage:

1. When to choose a file system? - large-scale data (images), provisional data (core files), low density information (archive files, logs).
2. When to choose a database? - simultaneous access, access to some details only, multiple platforms used, multiple applications use the same data
3. When to choose relational databases? - complex attribute queries, large databases
4. Object oriented databases? - extensive use of links between data acquisition objects

5 Access control policy

In multi-user systems, different users may have access to different data (authorization issue). It's necessary to determine how users in the system authenticate themselves (prove authenticity). In general, it's necessary for each actor to determine the operations to which he has access and to which of the data (objects). Access control is a system operation. It has to be common and uniform for all systems.

Possible solutions:

1. **Global access table** - Specifies the rights for the relation (actor, object, operation). If the rights are not explicitly granted, access is denied.
2. **Access control list** - For each class, it determines access rights for the relation (actor, operation). (eg. guest list at a party)

3. **Capability** - It connects the actor with the relation (class, operation). Allows the user with certain abilities to access the object. (eg. entering a party with an invitation).

6 A global control flow mechanism

Questions: How is the order of operations or activities determined in the system? Does it allow more than one concurrent user interaction? How does the system expect external (user) interaction?

The choice of a control flow mechanism affects the interfaces of the subsystems: event-driven control flow requires event handlers and threads require mutual exclusion in critical sections.

3 control flow mechanisms are used:

1. **procedure-driven control** - operations are waiting for input when they need data from actor. It's difficult when using OO programming languages because the order of the inputs is difficult to determine from the code because they're scattered between several objects.

2. **event-driven control** - the main loop is waiting for an external event. At an external event, the forwarder passes the data to the relevant object. Difficult implementation of procedures with interaction in several successive steps.

3. **threads** - the system can create any number of threads, each corresponding to a different event. If the thread requires additional input, it retrieves (waits) from the specific actor. The most intuitive way to work. Difficult debugging due to the non-deterministic nature of the system.

7 Borderline cases

questions??

Handling borderline cases affects the interfaces of all the subsystems. Exceptions:

- **user error** - inadequate user input
- **hardware error** - network errors, file system errors
- **software error** - the system or subsystem contains an error. An individual subsystem can expect errors from other subsystems and must be protected against them.

8 Architectural patterns

A pattern is a documented solution of a reoccurring problem. Solution is tested many times and generally accepted. The solution provides a way to solve a problem and not a direct solution that depends on specifics of each individual problem being solved.

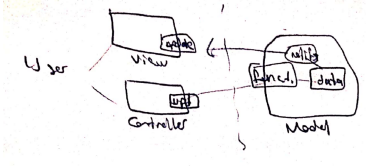
The term pattern is used in architectures of software systems: Architectural patterns or architectural styles. Each architectural pattern enables designing flexible system architectures that consist of components that are as independent as possible.

Patterns to be presented: 7

API - application programmer interface

1. **Layers** - each well designed layer has high cohesion. Coupling between layers is low. The lower layers don't deal with the specifics of the higher layers. The only communication is through the API. Knowledge of the other layers isn't required. Each layer can be designed independently. The layers can be independently tested. The lower layers can be designed generically and are reusable.

2. **Model/View/Controller** - the purpose is to separate the user interface layer from other parts of the system. The model combines all domain knowledge and operations, including objects whose properties we want to display to the user or control through user interaction. The view provides information to the user. The controller controls user interaction and transmits user data to the model and view. The controller collects user input and sends it to the model int he form of messages. The model maintains a central data structure. The view shows the state of the model. It's notified of any change to the model using the "unsubscribe/notify" protocol. The three components can be designed relatively independently. Communication between components is minimal. Ability to test the application (model) without the user interface (view+controller).



3. **Shared repository** - it enables central data management. Multiple subsystems access data from a shared warehouse. The subsystems are relatively independent and communicate only through a central data structure. The control flow can be dictated by the central repository (trigger) or by subsystems (synchronization with the repository lock). It's typical architecture of database management systems. The operation of each subsystem depends on the data managed by the common repository. All data is stored in a common central repository. All subsystems (application components) are dependent on this repository and access and affect its data.

4. **Microkernel** - it allows different versions of applications to offer different set of functionality or to differ from others by specific properties (eg. user interface). Despite the differences between the versions, all of them use the common functional kernel - microkernel. Different versions of the application should be built by extending the common function kernel.

5. **Client/Server** - A distributed system in which the server subsystem provides services to the client subsystems. The client request is usually sent via a remote procedure call mechanism or a common object broker, such as CORBA or Java RMI. The control flow between clients and servers is independent, with the exception of query / result synchronization on servers. Clients query the services of one or more servers. Servers do not know clients. The "client / server" architecture represents a generalization of the "Shared repository" architecture.

6. **Peer-to-peer** - it is a generalization of the "Client / server" architecture in which each subsystem can act as a server and as a client. Each subsystem accesses the services of others and offers its own services. The control flow of the subsystems is independent, with the exception of accesses synchronization. Designing peer-to-peer systems is more difficult than client / server systems because of the deadlock threat.

7. **Pipes and filters** - Subsystems process the data, which they obtain from other subsystems through a series of inputs, and pass it on to other subsystems via a series of outputs. The subsystems are called filters, the associations between them are called pipes. The architecture is variable - it is possible to change filters or change the configuration to achieve other goals. The relatively simple format of the data stream is transmitted through a series of filter subprocesses: each filter processes the data in its own way; the architecture is very flexible (almost any filter can be omitted; it's possible to replace the filter; new filters may be added; it's possible to change the order of filters). It's a suitable architecture for systems requiring data flow transformation without user intervention. It's not suitable for systems with more complex component interactions. The filters can be designed independently and have functional cohesion. The same filters can be used multiple times for multiple applications. Possible independent testing of individual filters.

9 Documenting

A good **architectural model** represents the system from multiple views: 5

1. logical division into subsystems
2. interfaces between subsystems
3. dynamics of interaction between components during run time
4. common subsystem data
5. deployment of components / subsystems to devices (HW)

Modeling begins with sketching an outline of the architecture depending on basic requirements and use cases, determination of basic system components, selection of architectural patterns used. It's useful for several independent groups to prepare the initial drafts separately and then combine them and apply the best ideas.

Improving architecture involves: 4

1. Identification of component interactions and determination of associated interfaces.
2. Distribution of data and functionality between system components.
3. Identifying options for reusing existing frameworks or deciding to build a new framework.
4. Consideration of each use case and customization of the architecture for its implementation.

9.1 UML description of an architecture

All UML diagrams can be useful for describing different aspects of an architectural model. Particularly important UML diagrams for architectural models are package diagrams, component diagrams and deployment diagrams.

9.2 Software architecture plan

why it's important to document the architecture of the system:

Documenting architecture contributes to better architectural solutions:

1. Documentation forces important decisions to be made before implementation begins.
2. Enables reviews of the architecture and thus offers the potential for further improvements.
3. Facilitates / enables / presents communication with those who will build the system; with those who will change the architecture of the system in the future; with those who will build other systems that'll connect to the planned one.

9.3 Architectural plan structure

5

1. **Purpose** - Which system or subsystem the architectural design refers to. Reference to requirements that are met (realized) by the planned system.
2. **Priorities considered** - A description of the design goals and priorities that were used in the design of the system.
3. **Summary of the Architectural Plan** - A general high-level description of the architectural plan, with which the user becomes familiar with the proposed solution.
4. **Description of the proposed architecture** - Identify the main challenges that need to be addressed. Description of alternatives and decisions on the chosen alternative, explaining the choice.
5. **Other details of the architectural plan** - Details that have not yet been mentioned and are relevant to the reader.

10 Architectural plan recension

X

In contrast to the recension of the SW requirements specification, the recension of the architectural plan does not include external reviewers (clients, users ...). The review should be done by developers who were not involved in the design. Reviewers should have sufficient interest in the in-depth study of the proposed solution.

1. An architectural model is **correct** if it permits the mapping of the requirements specification model to the system model (Can each subsystem be linked to use cases or nonfunctional requirements? Can each use case be linked to a group of subsystems? Can design goal be linked to non-functional requirements? Are all non-functional requirements met? Is there an access control method for each actor?)
2. Architectural model **completeness**: Does the model meet all the requirements? Were borderline cases taken into account? Does the model meet all defined use cases? Are all aspects of architectural planning addressed: HW / SW connections, data storage, access control, borderline cases) Are all subsystems defined.
3. Architectural model **consistency**: Are there any contradictions? Are conflicting design goals prioritized? Do any of the design goals conflict with the nonfunctional requirements? Do multiple subsystems or classes have the same name?
4. Architectural model **reality**: Can the system be developed? Are new technologies or new components included? Are they appropriate and robust enough for the designed system? Have performance and reliability requirements been analyzed? Have concurrency needs been taken into account?
5. Architectural model **readability**: Can the model be understood, even for people who have not been involved in modeling? Are the names of the subsystems clear? Do objects (subsystems, classes, operations) with similar names describe similar phenomena? Are all objects described with the same level of details?

Component design

teodoraristic

December 2021

1 Introduction

1.1 Object oriented and structural approach

Object oriented approach consists of system design (decomposition into subsystems) and component/object design (selection of implementation programming language and selection of algorithms, definition of data structures).

Structural analysis and design consist of **preliminary design** (decomposition into subsystems and definition of data structures) and **detailed design** (selection of algorithms; finishing definition of data structures; selection of implementation programming languages) - it's usually joined with preliminary design in a single phase.

We focus on object oriented approach of object design. Component design isn't algorithmic. The result is influenced by skills (knowledge, experience, creativity...). **Object design steps**: 4

1. using existing components
2. specifying interfaces
3. system restructuring
4. optimization

Component selection:

1. selection of existing solutions (off-the-shelf class libraries, frameworks and components)
2. customization of existing libraries, frameworks and components (changing the API when source code is available; adapter or bridge design pattern if source code isn't accessible).
3. designing new components based on the specified software architecture.

Communication between objects - objects communicate by passing messages. The messages consist of the names of the service called by the caller object from the called object and information needed to perform the service. In practice, messages are implemented in the form of procedure calls (name = procedure name; information = parameters).

2 Object design guidelines 4

Object oriented planning uses different approaches: the use of inheritance, delegation, abstraction/wrapping and modularity.

2.1 Inheritance (1)

Inheritance is used to achieve two goals:

1. **ordering of classes - taxonomy**: it's used in the requirements analysis phase to identify objects that are hierarchically independent. It helps to understand the models. **Inheritance detected by specialization** and **inheritance detected by generalization**.
2. **reuse** - it's used in the design phase. It contributes to the ability to use existing objects and to modify and extend them. **Extension of functionality (implementation inheritance)** or **interface specification (specification inheritance)**.

2.1.1 Implementation inheritance

It consists of using similar class with similar functionality. For example, an existing class List can be used to build a new class Stack. New methods Push(), Pop(), Top() will be built on existing functionality of methods Add() and Remove() in List. Methods of the basic class are available in the new class that inherits from the basic one.

2.1.2 Specification inheritance

Inherited interface. The subclass implements methods defined by a parent class that defines the interface. It enables different implementations with the same interface - external functionality.

2.1.3 Inheritance for reuse

1. **implementation inheritance** - the goal is to extend functionality while using the existing one
2. **specification or interface inheritance** - the goal is to define an interface for using different implementations of required functionality ; the subclass implements methods defined by a parent class that defines the interface

2.2 Delegation

Delegation is a way towards component reuse. A request from the client is received by a receiver that passes it modified to the delegate. The receiver makes sure that the delegate is properly used preventing inappropriate usage.

2.3 Delegation vs inheritance

1. delegation:
   (a) + flexibility; it's possible in all programming languages
   (b) - reduced performance due to encapsulation
2. inheritance:
   (a) + simplicity; it's supported in many programming languages; it's easy to add new functionality
   (b) - exposure of parent class details; any change in the parent class requires a change in the inherited class

2.4 Minimization of dependency

Software elements are dependent whenever a change of one SW element requires the change of some other software element. Dependencies are the main source of difficulties during maintenance. Software design leans towards minimized dependencies. **Measure of software dependencies is called Connascence**.

Examples of Connascence:

- **dynamic connascence**:
  1. **Connascence of Execution (CoE)** – e.g., required initialization before use
  2. ~~**Connascence of Timing (CoT)** – e.g. the irradiation device shutdown command must follow the device activation command in less than 50 milliseconds.~~
  3. **Connascence of Timing (CoT)** – e.g. the irradiation device shutdown command must follow the device activation command in less than 50 milliseconds.
  4. **Connascence of Identity (CoI)** - e.g. if two objects (O1 and O2) have to refer to the same O3 object (eg with pointers)
  5. **Diversity** - e.g. if class C inherits from classes A and B, then A and B must not contain methods of the same name.
- **static connascence** - Connascence of Name, Type, Meaning, Position, Algorithm.

2.5 Liskov substitution principle

It was named after Barbara Liskov and her lecture "Data abstraction and hierarchy" in 1987. If class **S is a subclass of class T**, then objects of class T may be replaced with objects of class S without altering any of the desirable properties of the program (correctness, task performed, etc.). If the base class meets the requirements of the application, then the LSP replacement class must also meet the same requirements so that from the interface point of view it works exactly the same and produces the same results for the same arguments.

3

2.6 Abstraction

→makes it easier to introduce change; the solution can be reused, requirements are easily traceable

It indicates the essential properties of an object that sets it apart from other objects. It defines the conceptual boundaries of objects, while allowing different interpretations depending on the view of the observer.

2.7 The open/closed principle

Keeps the class open for extensions adding operations or data structure fields. The class should be closed for changes if available for use by other classes. It must specify a stable and well-defined interface. In object oriented programming languages, this is realized through inheritance and polymorphism.

2.8 Encapsulation

Encapsulation is the process of wrapping elements by defining an abstraction that determines the structure and behavior of the elements. Encapsulation is used to separate "contract" interfaces (defined by abstraction) from implementation.

2.9 Class cohesion

Class cohesion measures the interconnectedness of the methods of the class's external interface. It shows how the class works in terms of implementing abstraction. We want high cohesion. There is no quantitative measure of class cohesion. Symptoms of low cohesion:

1. A class contains some components (methods) that are not defined for all class objects. Possible improvement with separation into two classes.
2. The class contains components (methods) that are not relevant to the purpose of the class (domain).

Inheritance can often be replaced by the use of roles. Example on page 33.

4

2.10 Modularity

The modular program consists of well-defined, conceptually simple and independent units - modules that communicate through well-defined interfaces. Advantages of modularity: 6

- easier to understand and interpret
- easier documentation
- easier modification
- easier testing and debugging
- better reusability
- easier to set up

A **module** is any unit of a program that consists of several parts (program, subroutine, package, class, operation, line of code).

Modularity principles: 5

1. **Small modules** - Designing with small modules is better.
2. **Information hiding** - Each module should hide its internal structure and processing to the other modules.
3. **Minimum privileges** - Modules should not have access to unnecessary resources.
4. **Coupling, connascence** - Dependence between the modules should be minimized.
5. **Cohesion** - Cohesion (internal dependency) should be maximized.

3 Design patterns

Design patterns are an elegant solution that's not evident for beginners. It's independent of system, programming language or application domain. It's extremely successful in real OO systems. It's simple and contains only a few classes. It's reusable and documented in a general way. It's object oriented, so they use OO mechanisms such as classes, objects, generalization and polymorphism.

Design patterns are defined by:

1. **pattern name**
2. **problem** - design samples contain a description of the problem for which the sample is appropriate
3. **solution** - design patterns give the elements and their relationships that form the solution to the problem
4. **implications** - results and consequences of using a design pattern

3.1 Classification of design patterns

According to the purpose:

1. **creational** - they define the process of creating objects
2. **structural** - they determine the composition of classes and objects
3. **behavioral** - they determine the way in which classes interact and the distribution of responsibilities

3.1.1 Creational design patterns 5

1. **Abstract Factory** groups object factories that have a common theme
2. **Builder** constructs complex objects by separating construction and representation
3. **Factory Method** creates objects without specifying the exact class to create
4. **Prototype** creates objects by cloning an existing object
5. **Singleton** restricts object creation for a class to only one instance

3.1.2 Structural design patterns 7

1. Adapter allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class
2. Bridge decouples an abstraction from its implementation so that the two can vary independently
3. Composite composes zero-or-more similar objects so that they can be manipulated as one object
4. Decorator dynamically adds/overrides behaviour in an existing method of an object
5. Facade provides a simplified interface to a large body of code
6. Flyweight reduces the cost of creating and manipulating a large number of similar objects
7. Proxy provides a placeholder for another object to control access, reduce cost, and reduce complexity

3.1.3 Behavioral design patterns 11

1. **Chain of responsibility** delegates commands to a chain of processing objects.
2. **Command** creates objects which encapsulate actions and parameters.
3. **Interpreter** implements a specialized language.
4. **Iterator** accesses the elements of an object sequentially without exposing its underlying representation
5. **Mediator** allows loose coupling between classes by being the only class that has detailed knowledge of their methods

1

0

6. **Memento** provides the ability to restore an object to its previous state (undo)
7. **Observer** is a publish/subscribe pattern which allows a number of observer objects to see an event
8. **State** allows an object to alter its behavior when its internal state changes
9. **Strategy** allows one of a family of algorithms to be selected on-the-fly at runtime
10. **Template method** defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior
11. **Visitor** separates an algorithm from an object structure by moving the hierarchy of methods into one object

3.1.4 Depending on the scope:

1. **class** - The pattern deals with classes and their interrelationships, which are determined by inheritance and are static
2. **object** - The pattern deals with relationships between objects that are dynamic and can change at runtime

3.2 Abstraction factory

The essence of the Abstract Factory Pattern is to "Provide an interface for creating families of related or dependent objects without specifying their concrete classes." Categories: **purpose: creational** and **scope: objects**. We want to avoid hard-coded widgets of the interface.

Abstract factory usage - The pattern is used when the system has to be configurable with different product families. The family of dependent products is designed to be used together and this restriction has to be ensured. We want to provide a product library, but we only want to disclose the interface and not the implementations.

Abstraction factory participants:

1. **AbstractFactory** - Declares an interface for operations that create abstract product objects
2. **ConcreteFactory** - Implementing operations to create abstract product objects
3. **AbstractProduct** - Declares an interface for the product object type
4. **ConcreteProduct** - Specifies the product object to be created with the associated ConcreteFactory and implements the AbstractProduct interface
5. **Client** - Uses only interfaces declared with AbstractFactory and AbstractProduct classes

Abstract factory implications - Client isolation from actual class implementations. Clients manipulate instances through an abstract interface. Product class names don't appear in the client code. It makes it easy to swap product families - only by using the second ConcreteFactory class, which only appears once in the client code. It promotes product consistency across families. Extending with new products is difficult, requiring the extension of the AbstractFactory class and all ConcreteFactory classes.

3.3 Adapter

The adapter design pattern allows otherwise incompatible classes to work together by converting the interface of one class into an interface expected by the clients. Categories: **purpose (structural) and scope (class)**. We want to make it possible to use an existing class, even if its interface isn't appropriate.

Adapter usage - The pattern is used if we want to use an existing class but its interface doesn't match the one we need or if we want to build a reusable class that can be used by a wide variety of other classes (including unknown ones).

Adapter participants:

1. **target** - Defines the domain-specific interface used by the Client object
2. **client** - Collaborates with objects using the Target interface principle
3. **adaptee** - Specifies the existing interface that needs to be customized

4. **adapter** - Translates the Adaptee object interface to the Target interface

Adapter implications - Adaptee interface is adapted to suit the Target. An adapter can override properties of an Adaptee class if it inherits it. In case of object composition, the adapter can adapt several Adaptee classes at a time. A two-way adapter is possible.

3.4 Observer

The Observer design pattern specifies a mechanism for linking multiple objects so that all objects are notified of a change of one of the objects. Categories: **purpose: (behavioral) and scope: (objects)**. We want the number of dependent objects to be unlimited and can change dynamically.

Observer usage - The pattern is used in the following situations:

1. A one-to-many dependency between objects should be defined without making the objects tightly coupled.
2. It should be ensured that when one object changes state an open-ended number of dependent objects are updated automatically.
3. It should be possible that one object can notify an open-ended number of other objects.

Observer participants:

1. **subject** - Knows the open ended Observer objects, there can be as many as you like. It provides an interface for connecting and disconnecting Observer objects.
2. **observer** - Specifies an interface for informing objects about changing a Subject object.
3. **ConcreteSubject** - Contains a state that is relevant to ConcreteObserver objects. Sends information about the change when it occurs.
4. **ConcreteObserver** - Contains a reference to the ConcreteSubject object. Keeps a state that must be consistent with the state of the ConcreteSubject object. It implements an update interface to maintain a consistent state with the ConcreteSubject object.

Observer implications - Abstract and minimal association between Subject and Observer objects (subject isn't aware of the actual class of the Observer; subject and observer may belong to different layers due to poor connectivity) and support of dispersed broadcasting (no need for mutual knowledge; the message is sent to all interested observers; ability to dynamically add or remove observers). A seemingly innocuous subject change can trigger an avalanche of changes to observers and dependent classes. Poorly defined class dependencies can lead to false updates.

3.5 Design patterns - general

There's a large number of design patterns. Everyone can define/propose their own additional design patterns. The selected patterns shown are presented to give a better idea of the design patterns in general. Design patterns are useful and knowing them can make programming easier and it can improve quality.

4 Documenting

The result of component design activity is a component design or a detailed design. Component design has to define:

1. the internal structure of all components
2. their dependence (interfaces)
3. their structure (operations, attributes, inheritances, compositions)
4. processing methods
5. algorithms
6. data structures
7. properties (functional and nonfunctional)

4.1 Interface specification

3

1. **Syntax** - it defined the elements of the communication medium and how they're grouped into messages
2. **Semantics** - specify the meaning of messages
3. **Pragmatics** - specify how messages are used to perform tasks

Testing

teodoraristic

December 2021

1 Introduction

1.1 Quality assurance

Each step of the software process has to be completed by review or testing. In this phase, we determine if the step results meet the step's input requirements (verification) and that they comply with the specification requirements (validation) as a guide to assessing user compliance. Quality assurance begins early in the software process, from the very beginning stages. The intensity of quality assurance increases in the implementation phase (coding) with source code review and verification. The peak of testing activities is in the testing phase with validation and system testing.

1.2 Verification and validation

- **Verification** - the individual components are tested for compliance with the component specifications - the component design. The individual components are integrated into the software system and tested according to the system specifications - the system design/
- **Validation** - it's executed by following a pre-prepared plan that follows SRS. It tests the compliance of the system with the requirements (functional and non functional, also taking into account implicit or self-evident requirements).

2 Recensions/Reviews

They include presentation of expert opinion (assessment), judgement on some new work in terms of quality. In the software process, as the final part of each task, is intended for quality assurance or verification and validation. In the implementation step we're talking about code review (code inspection).

Recension is an activity in which one or more people systematically scan the documentation for the purpose of finding possible errors. A review involves a meeting with the authors, but the reviewers can also review it individually or separately. Reviews shouldn't be directed towards the author, they have to be constructive and positive.

The purpose of reviews:

- Ensuring the quality of a peer-reviewed product (eg source code). Viewers can effectively find various discrepancies, ranging from deviation of business processes to simple errors in source code.
- A teaching tool for developers to share knowledge of quality improvement techniques (eg source code), consistency, ease of product maintenance, and use of different approaches.

Positive reviewers approach:

1. Ask instead of gravel - E.g. instead of "You did not follow the standards here!" use "What is the reason for using the approach used?"
2. Avoid "Why" questions - E.g. instead of "Why didn't you follow the standards here?" use "What is the reason for deviating from the standards ..."
3. **Don't forget the praise** - It is also important to notice the positive, e.g. creative solutions. This lets the author know that we see his work more broadly than just a series of mistakes.
4. **The topic of the review should be the work and not the author** - The purpose of the review is to contribute to the quality and not to determine the attributes and abilities of the author

5. **Be aware that there is more than one possible (good) solution** - Although the author did something differently than you would, this is not necessarily wrong.

Positive approach from the authors:

1. **Be aware that the peer-reviewed work is not you** - Development is a creative process and it is normal for you to be attached to your work. However, reviewers do not want to tell you that you are not a good developer (or person). Their mission is to point out the shortcomings and possibilities of better solutions. Even if reviewers do not do their job well, do not take a defensive position and listen to constructive comments (and overhear any attacks).

2. **Make a list of things that reviewers will focus on** - Use the list to reconsider your work and correct any errors found. This will not only reduce the number of errors found, but also shorten the audit time (to the delight of all involved).

3. **Help maintain standards** - Offer to supplement standards (eg coding) for what has been agreed and not included in existing standards. In this way, the documented solution will help with the works and reviews that will follow.

2.1 Documenting reviews

For reviews that are a list of comments, follow these guidelines:

1. **Start with a resume and be positive** - This shows the author your disposition and softens the comments that follow. The comment should contain positive messages to the author, as every work contains something good, which is worth mentioning (even if it is only syntactic correctness). A positive attitude is essential.

2. **Use electronic mechanisms to record comments** - E.g. copy to word and use word functions for commenting. This way you are not limited by the length of the comments and you have the opportunity to clarify your position and make the comments easier to formulate in a question form.

3. **Make an agreement that the author does not need to answer all the questions** - Good reviews also contain thought-provoking questions such as: "Wouldn't it be better to use the X design pattern?". Such questions do not need to be answered as they are intended solely to stimulate thinking. Such issues do not have a significant impact on the existing solution but have a positive long-term impact on product quality.

2.2 Principles

Reviews should be made for the most important documents (requirements specification, system and component design plans, test plans) and source code. An effective review team should consist of 2 to 5 members and include experienced developers. Participants should prepare for the review meeting. Do not review in-completed documents. Avoid discussing how to make corrections. Don't rush - when revising the source code, it's a good speed of 200 lines per hour. The review should not exceed two hours in total and four hours per day. Revisions should be made if more than 20% of the work is changed. The management should not be included in the reviews as this has a negative impact on the openness of other reviewers. No one should be offended.

2.3 Reviews vs testing

Reviews and testing are based on different ways of verifying a product. Reviews allow you to find bugs that affect maintenance possibilities and performance. Testing makes it possible to look for errors in complex solutions and have clear consequences. The chances of error are smaller when both review and testing are used. Reviews should take place before any testing.

3 Testing

Testing is the process of executing a software with the aim of finding an error before passing it on to end users. The tester performs a series of test cases in order to destroy the proper functioning of the program. A good test case is one that is likely to reveal an error not yet discovered. A test is successful if a previously unknown error is found.

3.1 Testing principles

1. All test cases should be related to the requirements - The worst mistakes are those that cause make program not to fulfill the requirements.
2. Testing should be scheduled before it begins - Test planning can begin as soon as the requirements specifications are finalized.
3. When testing, Law 80-20 (Pareto principle) applies - 80% of all errors detected during testing originate from 20% of software components. These components need to be tested so much more thoroughly.
4. Comprehensive testing is not possible - Even for smaller software components, the number of possible paths for running it may be too large to test all the options.
5. Independent testers perform the most effective testing - The developer who made the software is not the preferred tester because of a conflict of interest.

3.2 Who performs testing

- **software developer** - he understands the system well; the main interest is the timely delivery of the product; it's constructive and gentle when tested against the system
- **independent tester** - he doesn't know the system and has yet to know it; the main interest is quality assurance; it's destructive and will look for ways to break the system

3.3 Testability

Testability is a property that tells how easy it is to test a program. Good software testability is one of the design goals. Testability depends on several other features of the software:

1. **Operability** - The SW can be run smoothly
2. **Observalibiliy** - the SW provides clear insight into the details of SW execution
3. **Controlability** - the degree to which testing can be automated and optimized
4. **Decomposability** - testing can be focused on a single module.
5. **Simplicity** - the simplicity of the SW allows for more thorough testing.
6. **Stability** - the rate of change during testing.
7. **Understandability** - understanding the operation of the SW, including documentation.

3.4 Testing methods

Structural testing (white box testing) is testing based on knowledge of the internal structure of the program. The goal is to ensure that each condition has been satisfied at least once and every part of the source code has been executed at least once. **Behavioral testing (black box testing** is testing based on the declared external properties of the system.

3.5 The overall software structure

When testing, check the overall structure of the software - all paths of the execution. Logical errors and false assumptions are inversely proportional to the probability of path execution. We often believe that a certain path is unlikely to be taken during execution. Reality often contradicts intuition. Typing errors occur at random. They're likely to be present on untested routes.

3.6 Basis path testing

A **basis path** is a path through a software that contains at least one new piece of program code or a new condition. At least one part of the path isn't contained in other independent routes. The number of independent paths depends on the **cyclomatic complexity** - a measure of the logical complexity of the program.

Determining cyclomatic complexity V(G) based on execution graph (activity diagram) G:

- V(G) = numberof simple decisions + 1
- V(G) = number of contained areas of graph + 1 (page 30)

Basic paths define test cases for structural testing. They can be defined without using a graph, but graph makes it easier to follow the paths. Cyclomatic complexity can be determined by counting simple logical decisions. Complex decisions are considered to be two or more simple ones. Basic path testing is especially impotant for critical modules.

Industry research shows that the likelihood of failure is positively related to the cyclomatic complexity of V(G). Modules with higher V(G) are more subjected to errors.

Loops: simple loop, nested loop, related loop, unstructured loop.

Testing of simple loops - a set of test cases should include: leaving out the loop; a single pass through the loop; two passes through the loop; m passes through a loop where m¡n; n-1, n, n+1 passes through the loop. n is the maximum number of loop passes allowed.

Testing of nested loops - start with the innermost loop. All other loops should be executed with the least number of iterations. Test the loop for min+1, max-1 and the typical number of iterations. Move to one level a more external loop and test it the same way. Continue this way until you have completed the outermost loop.

Testing of related loops - if the loops are independent, then treat each one as a simple loop. If the loops aren't independent, treat them as nested loops.

3.7 Testing of conditions

Each decision is made based on some condition. Condition testing is based on the selection of such test cases to verify the correctness of the decisions against the decision parameters.

The following errors are possible:

1. Logical operator error (inappropriate, missing, redundant)
2. Logical variable error
3. Error in brackets of logical expression
4. Error in relational operator (¿, ¡, ¡=,¿ =, ==,! =)
5. Error in arithmetic operator

The test cases for testing the conditions partially overlap with the test cases for testing the basic paths.

3.8 Behavioral testing (black box testing)

Unlike white box, black box testing is used in the later stages of the testing process. It doesn't focus on the control flow, but the processed information. Test case planning is based on the following questions:

1. How to verify functional correctness?
2. How to check system behavior and throughput?
3. What input form good test cases?
4. Is the system sensitive to certain input values?
5. What are the boundaries of data classes?
6. What range of data can the system tolerate?
7. What impact do specific combinations of data have on the performance of the system?

3.9 Equivalence partitioning

Equivalence partitioning is a procedure for determining the classes of input parameters from which to select test data. An **equivalence class** is a group of valid or invalid input conditions. Depending on the input conditions, we define the equivalent classes as follows:

1. **The input condition is field-specified** - specifies one valid and two invalid equivalent classes.
2. **The input condition is a specific value** - it specifies one valid and two invalid equivalent classes.
3. **The input condition is determined by the membership of the set** - it specifies one valid and one invalid equivalent class.
4. **Boolean** - Specifies one valid and one invalid equivalent class.

3.10 Boundary value analysis

Boundary value analysis leads to the selection of test cases based on boundaries of input data. Testing with a BVA isn't only about the input but also the output. Other behavior testing techniques: error guessing methods, decision table techniques, cause-effect graphing.

4 Testing strategies

Several testing strategies have been proposed in the literature. They have the following characteristics in common:

- Testing is from the bottom up. It starts at the component level (eg classes, objects) and continues with the consideration of ever larger modules and ends with testing of the entire system.
- Different techniques are suitable for different stages of testing. Typical white-box testing takes place before behavioral (black-box) testing.
- Testing is done by the SW developer and (especially for larger projects) an independent test team.
- Testing and debugging are different and separate activities. Debugging must also be included in the test strategy.

4.1 Independent Test Group

Typically, a developer wants to demonstrate the quality of their product through tests (absence of defects, compliance with requirements ...). The purpose of testing is to look for defects and not to demonstrate quality. The developer isn't the preferred test person because there's a conflict of interest; a detailed knowledge of the system means a focused look at the system that's significantly different from the user's thinking models. The independent testing group doesn't have these limitations. However, the developer needs to work with the testing group to assist in setting up the environment and correcting any errors found.

4.2 Testing strategies

4.2.1 Unit testing

It covers the testing of individual components or units. Testing is based on source code (structural testing / whitebox). Typically, a unit test is the responsibility of the developer. The performance of the test depends on the experience of the developer. To test the unit, prepare a test driver. If the unit uses other units, they should be replaced as much as possible by stubs (minimum replacement implementation of the unit). Unit testing environment page 49.

4.2.2 Integration testing

Integration testing is intended to test a group of components that are integrated into a system or subsystem. The responsibility of the developers or (better) independent test team. Testing is based on system specifications (requirements, system design) - behavioral testing (black-box). The main problem is error localization, which can be solved by incremental integration. Integration testing strategies: the big bang approach and incremental construction strategy.

4.2.3 Incremental integration

1. **Top-down integration** - first, the high-level components which are connected to the low-level component stubs. The stubs have the same interface but very limited functionality. Higher level modules are tested with stubs. The stubs are one after the other replaced with full functionality (depth first). At integration of additional modules a subset of tests has to be performed again.

2. **Bottom-up integration** - first the integration of low-level components with full functionality, then the development and integration of higher-level components until the system is complete. Drivers are one after the other replaced with the full functionality (depth-first). Modules are grouped in clusters and integrated.

In practice, a combination of both approaches is used.

4.3 Testing of interfaces

It's used when subsystems are integrated into larger systems. The purpose is to find the errors of the interfaces and the errors in the assumptions related to the interfaces. It's also useful in OO approach, because objets are defined by their interfaces.

4.4 Validation testing

Validation testing is intended to determine whether a software product is performing in a manner that meets the "reasonable" expectations of users. What is "reasonable" is determined by the requirements specifications. There are two possible results: properties meet the requirements and deviations found and documented in a document in the deviation list. Deviations often need to be negotiated with the customer (eg due to time constraints).

An important part of validation testing is configuration review. This ensures that all elements of the software product are properly developed, cataloged and fit for the maintenance phase.

Alpha and beta testing:

1. The developer / tester is often unable to know in detail how the software product is used by the user, so it is appropriate to include end users in the final validation.

2. In the case of individual contracting authorities, the final validation shall be in the form of acceptance tests. These may be informal (test-drive) or formal with actual systematic test cases.

3. When there are many customers, the final validation takes the form of alpha and beta testing.

4. Alpha testing is performed on the developer's side in a controlled environment in the presence of users.

5. Beta testing is performed by one or more clients independently and without the presence of the developer.

4.5 System testing

System testing is intended to determine the behavior of a software product in the presence of other system elements (hardware, people, information). System testing involves a number of different tests designed to "fully test" the computer system. Different tests identify different properties of the system:

1. **Recovery testing** - restoring normal operation after the presence of faults
2. **Security testing** - checking the mechanisms of protection against unwanted interventions
3. **Stress testing** - operation of the system under excessive loads
4. **Performance testing** - system throughput testing

4.6 Stress test

Stress testing is testing the system at a load greater than the maximum planned. It often shows undetected errors. Excessive loading causes errors which shouldn't be catastrophic and there shouldn't be any tampering or loss of data. It's particularly important in distributed systems.

5 Debugging

Debugging is a diagnostic process as a result of successful testing. The purpose is to correct the errors found. The manifestation of the error (symptom) and the cause of it do not necessarily have an obvious interconnection. Debugging is a thought process that connects a symptom to the cause of an error.

5.1 Causes and symptoms

Symptoms and causes of errors may be geographically distinct. The symptom may disappear due to the elimination of another error. The reason for the symptom may be a combination of non-errors. The error may be due to a system error or a code compilation. The reason may lie in the assumptions that everyone believes. The symptom may be volatile.

Debugging techniques: **extensive testing, tracking, induction, deduction.**

Using tools (eg. debugging) to get a better idea of what's happening in the software. After eliminating the error, remember to perform regression testing.

6 Documenting

IEEE 829 is a Standard for Software Test Documentation and defines a set of recommendations for documenting software product testing. IEEE 829 identifies eight different documents covering three testing steps: preparation for testing, testing execution and completion of testing. IEEE 829 is a good starting point for determining test documents. However, the documents must be adjusted/modified to meet the needs of the project.

6.1 IEEE 829 - Preparation for testing

1. **test plan** - It defines how the test will be conducted: what will be tested, who will test, how will it be tested and when will it be tested.
2. **test design specification** - A detail of the test conditions and the expected outcome. This document also includes details of how a successful test will be recognized.
3. **test case specification** - A detail of the specific data that is necessary to run tests based on the conditions identified in the previous stage.
4. **test procedure specification** - A detail of how the tester will physically run the test, the physical set-up required, and the procedure steps that need to be followed.
5. **test item transmittal report** - It states what has been submitted to the testing process.

6.2 IEEE 829 - Testing execution

Test log - A detail of what tests cases were run, who ran the tests, in what order they were run, and whether or not individual tests were passed or failed.

Test incident report - A detail of the actual versus expected results of a test, when a test has failed, and anything indicating why the test failed.

6.3 IEEE 829 - Completion of testing

Test summary report - A detail of all the important information to come out of the testing procedure, including an assessment of how well the testing was performed, an assessment of the quality of the system, any incidents that occurred, and a record of what testing was done and how long it took to be used in future test planning. This final document is used to determine if the software being tested is viable enough to proceed to the next stage of development.

6.4 Test plan - general

A **test plan** is a document that defines a complete set of system test cases and additional information about the test process. The absence of a test plan means ad-hoc testing, which often results in poor quality software products. The test plan should be prepared well in advance of testing. The preparation of the test plan may commence as soon as the specification of the requirements is completed.

6.5 Test cases

The **test case** is an explicit set of instructions for detecting specific errors in the software system. The test case is used for a large number of tests. ~~The test case is used for a large number of tests~~.

The test case documentation contains:

1. **identifier and classification** - test case number and descriptive name. Indication of the system, subsystem or module to which it relates. Indication of the importance of the test.
2. **directions** - Accurate instructions to the examiner for what to do. Typically he does not refer to other documents.
3. **expected results** - Description of what the system is supposed to do as a result of running a test case. The tester reports an error if the system response does not match the expected one.
4. **cleaning (if necessary)** - Tells the tester how to bring the system back to 'normal' after the test.

6.6 Reports

Test results may depend on the environment, so test reports should include an identification of the environment (operating systems, versions of databases, and related systems ...). The results may depend on other SW running concurrently with the system tested.

Maintenance

teodoraristic

December 2021

1 Introduction

1.1 Deployment and maintenance

Deployment - upon acceptance of the software product, the customer may perform an acceptance test. This can be done at the developer (factory acceptance test - FAT) or at the customer (site acceptance test - SAT). The whole software system is delivered to the customer (executable software, documentation and data required for operation). This is followed by installation and transition to operation.

Operation and maintenance - During operation the user documentation must be available, in addition user training and user support may be provided. The developer maintains software products during the operation. Configuration management is important for the effective maintenance of software products.

1.2 Documentation and training

The documentation is intended for 2 target groups: **system maintainers** and **system users who daily use the system.** User training can be adapted to the specific needs of users or general (in the case of consumer products).

1.3 Customer support

Customer support is very important for users. Providing customer suppoert can be expensive (customer support service has to be established).

1.4 Installation

System installation is an organizational process in which an existing system is replaced by a new one. There are 4 installation strategies:

1. direct installation - when the new system starts, the old system stops.
2. parallel installation - the old and new system work in parallel until management decides to stop the old system
3. installation at a specific location - the new system is installed in a single starting location. It's up to the site to decide if and how the system will expand to other locations throughout the organization
4. phase installation - The transition from the old to the new system is gradual. It begins with the replacement of one (or several) functional components, and then the components are gradually replaced until the new system is fully installed.

Installation planning - what needs to be considered:

1. Data conversion of existing system (data error correction and transferring data from an existing system)
2. Planning to stop the existing system and switch to the system
3. Consideration of switching in the business process of an organization

1.5 Success factors

The most important measure of the success of a SW product and its development is the extent of its usage. The use of the system is influenced by a number of factors: 5

1. Personal interest of users
2. System features
3. Demographics (gender, age, race, income, education, etc.)
4. ~~Demographics (gender, age, race, income, education, etc.)~~
5. System performance
6. User satisfaction

1.6 Finalizing the project 5

1. Evaluation of the project (reviews)
2. Evaluation of the project team - assignment of members to other projects
3. Informing all participants of project completion and transition to operation and maintenance
4. Completion (redemption) of contracts with clients
5. Formal closure of the project

2 Maintenance

Maintenance in general - Functional checks, servicing, repairing or replacing of components and all other activities that keep the product in operation.

Software products do not wear out or break down and do not require periodic maintenance in that manner. Maintenance means changing a software product after release for the purpose of: error corrections, performance improvements, adaptation of the product to the changed environment, adapting to new requirements.

Maintenance is crucial for maintaining the use of a (software) product!

2.1 Maintenance cost

Maintenance is expensive! Typically, the cost of maintenance is 2 to 100 times higher than the cost of development! The cost (difficulty) of maintenance is influenced by both technical and non-technical factors: application type, system uniqueness, personnel fluctuation and experience, system lifetime, variable environment dependency, hardware characteristics, design quality (structure), source code quality, documentation quality, testing quality, number of users, use of tools...

2.2 Changes of software products

In most cases, the starting point for maintenance is the existing product. Maintenance includes 4 types of software changes:

1. **corrective changes** - error correction (21%)
2. **adaptive changes** - adaptation to changed conditions (25%)
3. **perfective changes** - improvements of the software product (50%)
4. **preventive changes** - elimination of software product degradation (4%)

2.3 Evolution

Software maintenance is software evolution (the gradual change of something, usually into more complete, more sophisticated forms).

What is evolving:

1. **requirements specification** - requirement evolution
2. **design** - architecture evolution
3. **implementation** - data evolution; source code evolution; documentation evolution; evolution of technology used
4. **testing** - testing evolution (test cases)

Evolution must be synchronous: co-evolution problem.

2.4 Evolutionary types of software products

Meir Manny Lehman identified three types of software products according to the need for maintenance:

1. **S-system**: Formally defined systems that do not require change throughout their lifespan. They rarely occur in practice!
2. **P-system**: systems whose requirements are based on an applied solution to a problem. They need incremental adaptation to the real world.
3. **E-system**: systems embedded in the real world require constant adaptation to the real world. Most software products belong to this group!

Lehman's laws of software evolution:

1. **Law of continuing change** - Systems must be continuously adapted or they become progressively less satisfactory to use. The variance between the system and its operational context leads to feedback pressure forcing change in the system.
2. **Law of continuous growth** - Functional capability must be continually increased over a system's lifetime to maintain user satisfaction. In any system implementation, requirements have to be constrained. Attributes will be omitted, these will become the irritants that trigger future demand for change. Feedback from the users.
3. **Law of increasing complexity** - As a system evolves, its complexity increases unless work is done to maintain or reduce it. If changes are made with no thought to system structure, complexity will increase and make future change harder. On the other hand, if resource is expended on work to combat complexity, less is available to system change. No matter how is balance is reconciled, the rate of system growth inevitably slows.
4. **Law of declining quality** - Unless rigorously adapted to meet changes in the operational environment, system quality will appear to decline. A system is built on a set of assumptions, and however valid these are at the time, the changing world will tend to invalidate them. Unless steps are taken to identify and rectify this, system quality will appear to decline, especially in relation to alternative products that will come onto the market based on more recently formulated assumptions.

Validity of Lehman's laws - Lehman's laws fit well with commercial software. In case of non-standard software, they may not be valid. eg. Linx (open source software); because developers are solving their own problems, there's less time pressure
developers aren't compensated for open-source sw

2.5 Maintenance difficulty

There's a gap between a need for change (error correction, customization) and the need for the availability of the system for users (tendencies for minimal and quick implementation of changes and patches). The decision has to be made between fast, messy problem solving and in-depth and elegant solutions.

2.6 The end of maintenance

If we stop maintaining the software, it begins to decay. The decision to evolve or discontinue maintenance is related to the following questions: 7

1. Is the maintenance cost too high?
2. Is the system reliability insufficient?
3. Is the system no longer able to adapt to changes fast enough?
4. Is the system performance insufficient?
5. Are the system's functionalities of limited usefulness?
6. Can other systems do the same job better, faster, cheaper?
7. Is the cost of maintaining the hardware high enough to warrant replacement with a new, cheaper one?

2.7 Who does maintenance

basically this ↴

There's a separate maintenance team: it can be more objective; it might be easier to distinguish between how the system should work and how it actually works; it enables members of the old team to focus on new projects. Maintenance is performed by a part of the development team. The system will be built to make maintenance easier. Team members may feel overconfident and neglect documentation that helps with maintenance.

2.8 Maintenance process 8

1. Obtaining maintenance applications /requests/ ideas
2. Specifying a change request
3. Designing the change
4. Implementing changes
5. Testing
6. Documenting
7. Training
8. Deployment

Maintenance techniques and tools:

1. Configuration management
2. Impact analysis
3. Engineering, reverse engineering, re-engineering

2.9 Engineering

1. **Forward engineering** is a traditional process that runs from a high level of abstraction and logical, execution independent design to the physical execution of the system.
2. **Reverse engineering** is the process of analyzing a system to identify system components, their interactions, and represent the system in a different form with a higher degree of abstraction.
3. **Reengineering** is the exploration and conversion of a system to reconstruct and implement it into a new design with the same external functionality.

3 Rationale

Rationale is the explanation of the logical reasons or principles employed in consciously arriving at a decision or estimate. Rationales usually document why a particular choice was made; how the basis of its selection was developed; why and how the particular information or assumptions were relied on and why the conclusion is deemed credible or realistic.

Rationale: A documented and reasoned decision that determines how a product is developed in its software process.

Rationale includes:

1. **Issues** to which it relates
2. **Alternatives** considered
3. **Decisions** taken to resolve the issue
4. **The criteria** that led to the decision
5. **Discussions** that were needed in the development team to reach a decision

Agile development

teodoraristic

December 2021

1 Managing uncertainty

Software development is a learning process to reduce uncertainty:

1. **Project scope** - determine the functionality
2. **Resources required** - development team members (time), equipment needed
3. **Meeting the deadline** - product delivery time

1.1 The traditional approach

A typical example is a linear programming process. We want to manage the uncertainty as early as possible in the project:

1. by describing the system functionality in the analysis phase
2. defining structure of the system in the system design phase
3. define the structure of modules (programs, classes) in the detailed design phase
4. and when the uncertainty is minimal we start coding.

Uncertainty is reduced gradually in each phase and by each product of each phase (SRS, design plan, component design plan...) Reduced uncertainty contributes to easier implementation, better solutions and easier maintenance.

1.2 Traditional difficulties

The development of virtually every software product faces a series of uncertainties that are difficult to completely eliminate in the traditionally defined phases of a project: Requirements are only fully clarified in the design or even implementation or testing phase (inconsistencies, deficiencies detected). In each next phase difficulties of the previous one are detected.

The consequences:

- Products such as SRS, design plans etc. are incomplete and must be supplemented in later phases.
- The project scope, timetable, and costs vary as the project progresses!

1.3 Alternatives

Why extensive pre-planning and documentation if the right functionality and execution are only proven during coding and testing?

1. Informal implementation of the analysis and design phases – during coding!
2. Frequent and automatic testing during implementation!
3. Involvement of the customer / user in the development process!

1.4 Agile development

In agile approaches, some phases are performed informally in order to increase process flexibility and shorten the time to delivery (earlier return of investment, better cooperation with customer). Agile approaches replace formality with informal, open relationship within the development team; collaborating within and outside the team; more active quality control.

1.5 The values of agile approaches

1. The individual and the relationship between individuals is more important than the processes and tools.
2. Running software is more important than extensive documentation.
3. Customer involvement is better than contractual negotiations.
4. Responding to change is more important than strictly following a plan.

1.6 Principles of agile approaches

1. Divide functionality into small independent components
2. Frequent releases of new software versions
3. Changes to requirements are welcome, even if late
4. Daily cooperation of technical and business staff
5. Direct involvement of a customer representative in the team
6. Personal communication (four-eyed) is best
7. Projects should be based on motivated individuals who need to be trusted.
8. Simplicity is a value
9. The teams should get organized by themselves

1.7 Agile methodologies

1. **Extreme Programming (XP)**
2. **Scrum**
3. Dynamic Software Development Method (DSDM)
4. Feature Driven Development (FDD)
5. Adaptive Software Development (ASD)
6. Crystal Clear
7. Rapid Application Development (RAD)

1.8 XP - Extreme programming

Extreme Programming is a discipline of software development based on values of simplicity, communication, feedback, and courage. It works by bringing the whole team together in the presence of simple practices, with enough feedback to enable the team to see where they are and to tune the practices to their unique situation.

Extrema of extreme programming:

1. If code reviews are good, do them all the time (pair programming)
2. If testing is useful, let everyone test all the time ("unit tests")
3. If simplicity is useful, keep the system as simple as possible, as long as it meets the requirements.

4. If planning is useful, everyone should plan daily (continuous reordering).
5. If architecture is important, it should be planned and improved by everyone.
6. If integration testing is important, the integration and testing should be performed multiple times a day – continuous integration.
7. If small iterations are good, keep them really small (hours rather than weeks).

1.9 XP - Project execution

7

1. Defining stories - the attributes that a customer / user wants.
2. Estimate duration and cost for each story.
3. Choosing stories for the next release / version.
4. Define tasks of each next release.
5. Preparation of test cases for each task.
6. Programming in pairs. No specialization from developers. Presence of customer representative.
7. Continuous integration.

1.10 XP principles

12

1. **"Planning game"**: Setting the goals of the next version by considering business needs and technical assessments.
2. **"Small releases"** - A simple system is released and is upgraded with small and frequent updates (new releases / versions).
3. "Metaphor" - a simple story about the functioning of the final system to guide the development.
4. **"Simple design"** - the system must be designed as simply as possible (unnecessary complexity must be removed as soon as it is identified).
5. **"Testing"** - testing is happening all the time, "unit tests" are the basis for programming functionality.
6. **"Refactoring"** - continually rearranging the structure of a system to reduce complexity and avoid code duplication.
7. **"Pair programming"** - always two programmers at one computer.
8. **"Collective ownership"** - a common source code of the entire system that anyone can change at any time.
9. **Continuous integration** - changes are integrated into the system several times a day, after each task is completed.
10. **40-hour week** - never work more than 40 hours a week.
11. **"On-site customer"** - The customer representative is included in the development team in terms of continuous availability to answer developer questions.
12. **"Coding standards"** - strictly following the rules of programming, with an emphasis on how to communicate through source code.

1.11 Scrum roles

Scrum defines the skeleton of a software process in which:

1. **Scrum Master** ensures the smooth execution of the process (the role of project manager). NOT a team leader!
2. **Product Owner** represents the interests of the customer and takes care of the proper direction of the project (business, choice of functionality ...).
3. **Team**, a multidisciplinary team of 7 (+/- 2) members, whose task is to analyze, design, implement, test ...

External actors that influence the scrum process:

1. **Stakeholders** - customers, users - are the ones who will benefit from the project and justify its implementation. Sprint reviews are included in the process.
2. **Management** (of the organization) - which establishes the conditions for project implementation.

Pig roles are scrum master and team members. Chicken roles are product owner, management and stakeholders. A pig and a chicken are walking down a road. The chicken looks at the pig and says, "Hey, why don't we open a restaurant?" The pig looks back at the chicken and says, "Good idea, what do you want to call it?" The chicken thinks about it and says, "Why don't we call it 'Ham and Eggs'?" "I don't think so," says the pig, "I'd be committed, but you'd only be involved."

1.12 Scrum - the project course

The owner of the product arranges and takes care of the product backlog -wish list or features of the software product. Specify the release backlog as a subset of the product backlog. For each feature the execution time needs to be estimated. The release backlog is distributed among several (typically 4-12) parts, so that each part takes up to 4 weeks to complete. These parts are called "sprints" and the related parts of the backlog is called "sprint backlog".

1.13 Scrum - sprint

In each sprint, the team implements potentially deliverable product. The sprint backlog must not be changed during the sprint run (fixed requirements). At the end of the sprint, the team demonstrates a developed solutions.

1.14 Scrum - meetings

Declared meetings: 5

1. Sprint Planning Meeting
2. Daily Scrum
3. Scrum of scrums
4. Sprint Review Meeting
5. Sprint Retrospective]

Sprint Planning Meeting: 4

1. At the start of the sprint
2. Choosing the scope of work (what to do)
3. Preparing a Sprint Backlog with estimates of the required time (regarding the team)
4. The meeting is limited to 8 hours

Daily scrum:

1. Every day during the sprint period. 6
2. A quick standing meeting on the status of a sprint

3. Always starts at the same prescribed time and place.
4. Everyone is welcome, only the pigs have the word.
5. Meeting time limited to 15 minutes.
6. Each team member answers three questions: What did you do until yesterday? What are you planning to do today? Do you have any problems that hinder you from completing the task?

Scrum of scrums: 4

1. Every day after the daily scrum.
2. Meeting between representatives of several teams of the same project.
3. Each team is represented by one member.
4. The content is the same as in the daily scrum: What has your team done since our last meeting? What will your team do until our next meeting? Is there anything that hinders you from completing the task? Do you see the need for teams to work together?

Sprint Review Meeting: 3

1. Review of completed and unfinished / unfinished work.
2. Presentation of results to stakeholders (demo).
3. Four hours limit.

Sprint Retrospective: 4

1. All team members present their views on the completed sprint.
2. Care for continuous improvement of the software process.
3. Answer two questions: What went well in the sprint? What could be improved in the next sprint?
4. Three hours limit.

1.15 Scrum - burn down chart

Burn down chart - a publicly released graph showing the remaining work required to finish the sprint backlog. It's updated daily. It offers easy insight into sprint progress.

1.16 General on agile development

Agile approaches don't mean ad-hoc programming. They specify the standards to be followed: 4

1. coding style
2. a way to integrate new functionalities and changes
3. testing method (unit testing and integration)
4. higher level software process (requirements collection and functionality selection, releases)

The success of agile projects is affected by: 5

1. the size (more than 20 developers)
2. geographic distribution of the development team
3. management culture of the company
4. forced use of agile approaches
5. critical tasks where errors aren't allowed

Advantages of agile methods: 8

1. Customer satisfaction by rapid, continuous delivery of useful software.
2. People and interactions are emphasized rather than process and tools. Customers, developers and testers constantly interact with each other.
3. Working software is delivered frequently (weeks rather than months).
4. Face-to-face conversation is the best form of communication.
5. Close, daily cooperation between business people and developers.
6. Continuous attention to technical excellence and good design.
7. Regular adaptation to changing circumstances.
8. Even late changes in requirements are welcomed.

Agile approach criticism:

1. Insufficient planning.
2. Insufficient documentation.
3. Urgent above average programmers.
4. Difficult financial planning.
5. More difficult quality assurance.

Drawbacks of agile methods: 4

1. Extensive documentation is missing
2. Extensive testing is useful, but not everything is testable
3. Difficult to track fast changing requirements
4. Continuous code refactoring may introduce errors
5. **Extensive documentation is missing**

   The software product must be maintained: some software products have been around for decades. Even after the development is complete, we need knowledge of the structure of the system and knowledge of the reasons for the decisions made.

7. **Continuous code refactoring may introduce errors** - The modifications are meant to improve software performance and as code refinement. The "suboptimalities" of the code may be due to the specific specific requirements. Changing the software operation (algorithm change) cannot be automated, so it is subject to errors!
8. **Not everything can be tested.** - Building automated tests is useful, but tests are always limited. When is testing sufficient? Testing sequential programs is difficult, but what about in the case of parallelisms? How to (automatically) test the quality of security-critical applications?
9. **Difficult to track fast changing requirements** - How to manage inconsistent requirements? Why was the change made? Where does the change request come from? Was the change justified?

1.17 Continuous (des)integration

The success of integration depends on:

1. frequent updating of the source code
2. frequent product build
3. immediate testing
4. immediate debugging

1.18 Agile and formal methods

Agile approaches may be more optimal than formal (depending on project characteristics). Agile approaches cannot completely replace the formal approaches. Agile and formal approaches can be complementary.

1.19 Software processes in general

The processes are more or less adaptive or predictive. Agile processes are more adaptive, but they're more difficult to be predicted for the future. The processes are more or less optimized. Formal processes are more optimized, but they have more problems coping with changes.