Software Engineering*

October 8, 2014

Contents

1 Software lifecycles (academic)

1.1 Software release lifecycle

The software release life cycle is composed of discrete phases that describe the software's maturity as it advances from planning and development to release and support phases.

Pre-alpha: refers to all activities performed during the software project prior to system testing. These activities can include requirements analysis, software design, software development and unit testing.There are several types of pre-alpha versions: development releases, nightly builds, milestones, etc.

Alpha: in this phase, developers generally test the software using white box techniques. Additional validation is then performed using black box or gray box techniques, by another testing team. Moving to black box testing inside the organization is known as alpha release. Alpha software can be unstable and could cause crashes or data loss. The alpha phase usually ends with a *feature freeze*, indicating that no more features will be added to the software. At this time, the software is said to be feature complete.

Beta: in this phase, pilot testing is performed by a limited number of end users in the target environment. The users of a beta version are called beta testers. They are usually customers or prospective customers of the organization that develops the software, willing to test the software for free or for a reduced price.

Release Candidate: The term release candidate (RC) refers to a version with potential to be a final product, ready to release unless fatal bugs emerge. In this stage of product stabilization, all product features have been designed, coded and tested through one or more beta cycles with no known showstopper-class bug.

RTM: The term "release to manufacturing" or "release to marketing" (both abbreviated RTM), also known as "gold", is a term used to indicate that the software has reached a point that it is ready to or has been delivered or provided to the customer. RTM happens prior to general availability (GA) when the product is released to the public.

GA: General Availability or General Acceptance (both abbreviated GA) is the point where all necessary commercialization activities have been completed and the software has been made available to the general market

1.2 Development lifecycles

- Waterfall model: it's an activity-centered life cycle that prescribes sequential executions of subsets of the development processes and management processes
- v-Model: it's a modification of the waterfall model that makes explicit the dependency between development activities and verification activities.
- Spiral model: is an activity-centered life cycle model that was devised to address the source of weaknesses in the waterfall model, in particular to accommodate infrequent change during the software development. This model focuses on addressing risks incrementally, in order of priority.
- Unified Software Development Process: distinguishes important time ranges in software development called cycles. (birth, childhood, adulthood, retirement, death) Each cycle has four phases: Inception, Elaboration, Construction, Transition.
- Issue-based model: based on solving issues that are generated progressively.
- Agile software development: it is a group of software development methodologies based on iterative and incremental development, where requirements and solutions evolve through collaboration between self-organizing, cross-functional teams. e.g. Scrum

2 Requirements elicitation

2.1 Functional Requirements

Functional requirements describe the behavior and the interactions of the system with its environment independently of its implementation. The environment includes the user and any other external system with which the system interacts.

2.2 Nonfunctional Requirements

Nonfictional requirements describe aspects of the system that are not directly related to the functional behavior. Nonfictional requirements include a broad variety of requirements that apply to many different aspects of the system. Some nonfunctional requirements categories are: usability, reliability, performance, supportability, constraints (implementation requirements, legal categories,...).

2.3 Analysis

Once the requirement elicitation activity is over the development team focuses on the analysis phase. Analysis focuses on producing a model of the system, called *analysis model*, that is correct, complete, consistent and verifiable. The two are different since during this phase the developers focus on structuring and formalizing

the requirements elicited from users. Doing so leads to new insight and the discovery of errors in the requirements.

The analysis model is composed of three parts: the *functional model*, represented by use cases and scenarios, the *analysis object model*, represented by class and object diagrams, and the *dynamic model*, represented by state machine and sequence diagrams.

2.4 RAD

The *Requirement Analysis Document* (*RAD*) is the result of the requirements elicitation (and analysis) activities. This document completely describes the system in terms of functional and nonfunctional requirements. The first part of the document is written during the requirement elicitation phase while the second part is completed during the analysis phase.

3 Design

3.1 Design principles

- *Coupling* is the number of dependencies between two subsystems. If two subsystems are loosely coupled, they are relatively independent, so modifications to one will have little impact on the other. If two subsystems are strongly coupled, modifications to one subsystem is likely to have impact on the other.
- *Cohesion* is the number of dependencies within a subsystem. If a subsystem contains many objects that are related to each other and perform similar task, its cohesion is high. If a subsystem contains a number of unrelated objects, its cohesion is low.

When designing a system a good property is that subsystems are as loosely coupled as possible (*minimize coupling*) and as highly cohese as possible (*maximize cohesion*).

- *Layering* allows systems to be organized as a hierarchy of subsystems, each providing higher-level services to the subsystems above it by using lower-level services from subsystems below it.
- *Partitioning* organizes subsystems as peers that mutually provide different services to each other.

3.1.1 Different purposed for classes and inheritance

Class hierarchies and polymorphism can be used to model both the application domain and the solution domain.

- *Application objects*, also called *domain objects*, represents concepts of the domain that are relevant to the system.
- *Solution objects* represent components that do not have a counterpart in the application domain, such as persistent data stores, user interface objects, or middleware.

In the same way inheritance can be used both to classify analysis object into taxonomies or to reduce redundancy and enhance extensibility of the software product.

- *Specification inheritance* consists of the classification of concepts into type hierarchies.
- *Implementation inheritance*, instead, is the use of inheritance for the sole purpose of reusing code. Usually *delegation* (i.e. implementing an operation in a certain class by resending a message to another class) is a preferable mechanism to achieve reuse.

3.1.2 Kent Beck's four rules of simple design

- Runs all the tests.
- Expresses every idea that we need to express. (intent)
- Says everything OnceAndOnlyOnce. (no duplication)
- Has no superfluous parts. (minimize classes/methods)

3.2 Architectural styles

Model-View-Controller: View renders data, observing Model, and you can have multiple. Controller receives user input and makes modifications to Model.

Repository: Independent component interact only through the repository.

Multi-tier architecture: Each tier is dedicated to a purpose. Usually interface, application logic and storage.

Pipes and filters: Subsystems process data received as inputs and send results to other subsystems via a set of outputs.

Client-Server: the server provides services to a client which requires them.

Peer-to-peer: generalization of client server where subsystems are both.

3.3 Design patterns

In object-oriented development, *design patterns* are template solutions that developers have refined over time to solve a range of recurring problems.

3.3.1 Creational patterns (5)

These patterns have to do with class instantiation. They can be further divided into class-creation patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation to get the job done.

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

3.3.2 Structural patterns (7)

These concern class and object composition. They use inheritance to compose interfaces and define ways to compose objects to obtain new functionality.

- ***Adapter*** allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class.
- *Bridge* decouples an abstraction from its implementation so that the two can vary independently.
- *Composite* composes zero-or-more similar objects so that they can be manipulated as one object.
- *Decorator* dynamically adds/overrides behaviour in an existing method of an object.

- ***Facade*** provides a simplified interface to a large body of code.
- *Flyweight* reduces the cost of creating and manipulating a large number of similar objects.
- *Proxy* provides a placeholder for another object to control access, reduce cost, and reduce complexity.

3.3.3 Behavioral patterns (11)

Most of these design patterns are specifically concerned with communication between objects.

- *Chain of responsibility* delegates commands to a chain of processing objects.
- ***Command*** creates objects which encapsulate actions and parameters.
- *Interpreter* implements a specialized language.
- ***Iterator*** accesses the elements of an object sequentially without exposing its underlying representation.
- *Mediator* allows loose coupling between classes by being the only class that has detailed knowledge of their methods.
- *Memento* provides the ability to restore an object to its previous state (undo).
- ***Observer*** is a publish/subscribe pattern which allows a number of observer objects to see an event.
- *State* allows an object to alter its behavior when its internal state changes.
- *Strategy* allows one of a family of algorithms to be selected on-the-fly at runtime.
- *Template method* defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior.
- *Visitor* separates an algorithm from an object structure by moving the hierarchy of methods into one object.

4 Object Oriented Design

- *Aggregation* (vs *composition*):
  - In aggregation, A uses B. B exists independelty of A.
  - In composition A owns B. B has no meaninig without A (i.e. B is a component of A).
- *Class (vs object)*: A class is a written piece of code that defines the behavior of a given class. An object is an instance of a class obtained through a process of instantiation.
- *Class / static initializer*: block of code that is used to initialize static members. Similar to a "constructor for class/static methods/attributes".
- *Class / static method*/attribute: methods that don't belong to any object (i.e. instance) but to the class itself.
- *Constructor*: block of code that is used to inizialize new objects (i.e. class instances). It's executed every time a new instance is created.
- *Delegation / forwarding*: is a design pattern where an object, instead of performing one of its stated tasks, delegates that task to an associated helper object.
- *Dependency Injection (DI)*: A desingn pattern that removes hard-coded dependencies (i.e. services) from dependent objects (i.e. clients). The service is made part of the client's state. Passing the service to the client, rather than allowing a client to build or find the service. There are three main ways of injecting dependencies:

  - constructor injection: the dependencies are provided through a class constructor.
  - setter injection: the client exposes a setter method that the injector uses to inject the dependency.
  - interface injection: the dependency provides an injector method that will inject the dependency into any client passed to it. Clients must implement an interface that exposes a setter method that accepts the dependency.

- *Dependency Inversion Principle (DIP)*: tells us how to achieve decoupling in software. The principle states:
  - High-level modules (i.e. Client) should not depend on low-level modules (i.e. dependencies). Both should depend on abstractions.
  - Abstractions should not depend on details. Details should depend on abstractions
- *Destructor / finalizer*: a block of code that is executed every time an object is destroyed / deallocated.
- *Encapsulation*: boundling data and the functions operating on such data for the purpose of restricting access to some data or code (data hiding)
- *Inheritance*: A mechanism for code reuse to allow extension of the original software. Not to be confused with subtyping (Subtyping is described by Is-a relationship, while inheritance is pure code reuse).
- *Instantiation*: The process of creating an object from a class.
- *Interface / protocol (vs abstract class)*: both can't be instantiated but interfaces only specify which methods need to be implemented without any implementation while abstract classes provide some implementation and attributes.
- *Interface Segregation Principle*: no client should be forced to depend on methods it does not use. Once this principle is applied interfaces which are very large are split into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them.
- *Inversion of Control (IoC)*: An implementation of the dependency inversion principle. IoC achieves DIP by putting interfaces to low level modules inside high level modules. Low level code implements these interfaces. This "inverts" the traditional pyramid of OO where a dependency sits "inside" the client.
- *Is-a relationship (vs has-a relationship)*: is-a relationships are usually described by inheritance, has-a relationships are usually described by composition
- *Liskov Substitution Principle*: if S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e., objects of type S may substitute objects of type T) without altering any of the desirable properties of that program (correctness, task performed, etc.).
- *Method (vs function)*: methods have access to class attributes because the object is implicity passed to the method.
- *Multiple inheritance*: whenever a class is allowed to inherit from more than one class. This can generate problems like diamond inheritance.
- *Open-Closed Principle*: software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification; that is, such an entity can allow its behavior to be modified without altering its source code.
- *Overloading (vs overriding)*:
  - overloading is when you define two methods with the same name, in the same class, distinguished by their parameter list.
  - overriding is when you re-implement a method of the superclass in the subclass.

- *Polymorphism*: providing a single interface to different types. Several ways of achieving it:
  - function overloading (or *adhoc* polymorphism)
  - generics (or *parametric* polymorphism)
  - subtyping (*inclusion* polymorphism)
  - duck typing (whenever the language is dynamically typed)
- *Single Responsibility Principle*: every context (class, function, variable, etc.) should have a single responsibility, and that responsibility should be entirely encapsulated by the context.
- *SOLID principles*: first five principles of OO Design: Single Responsibility Principle, Open-Closed Principle, Liskov-Substitution Principle, Interface Segregation Principle, Dependency Inversion
- *Superclass or base class (vs subclass or derived class)*: a subclass inherits from a superclass
- *Virtual method* (pure virtual): a pure virtual function or pure virtual method is a virtual function that is required to be implemented by a derived class if that class is not abstract.

5 Verification and Validation (V&V)

Verification: (*Are we building the product right?*) The process of evaluating software to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. Often an internal process, static (doesn't require the code to execute), involves humans checking the code, the documents, the files (code inspection).

Validation: (*Are we building the right product?*) The process of evaluating software during or at the end of the development process to determine whether it satisfies specified requirements. Often a process that involves users (or automated tests), dynamic (requires the code to execute).

5.1 Component inspection (Verification)

The purpose of component inspection is to find faults in a component by reviewing its source code in a formal meeting. It is very time consuming but generally more effective than testing in uncovering faults. It is used as a complement to testing in safety-critical projects.

6 Testing

- A *fault* (also known as a *bug* or *defect*) is a design or coding mistake that may cause abnormal component behavior.
- A *failure* is a deviation between the specification and the actual behavior. A failure is triggered by one or more erroneous states. Note that not all erroneous states trigger a failure!
- A *test case* is a set of inputs and expected results that exercises a component with the purpose of causing failures and detecting faults.

6.1 Types of testing

- *Black box testing*: tests the functionality of the system without knowing the internal structure
- *White box testing*: a complete knowledge of the internal structure of the system is used during the testing process
- *Gray box testing*: some knowledge of the internal structure of the system is used to write better test cases but then the system is tested as a black box

6.2 Types of testing examples (more specific)

6.2.1 Unit testing

Unit testing focuses on the building blocks of the software system, that is, objects and subsystems. There are three motivations behind focusing on these building blocks: it reduces complexity of overall test activities, it makes easier to the pinpoint and correct faults and allows parallelism of testing activities.

- *Test case* is a set of conditions or variables under which a tester will determine whether an application, software system or one of its features is working as it was originally established for it to do.
- *Test suite* is a collection of test cases.
- *Test fixture* is a fixed state of the software under test used as a baseline for running tests (i.e. all the things that must be in place in order to run a test and expect a particular outcome).
- *Stub* is a piece of code used to stand in for some other programming functionality. A stub may simulate the behavior of existing code (such as a procedure on a remote machine) or be a temporary substitute for yet-to-be-developed code.
- *Mock objects* are simulated objects that mimic the behavior of real objects in controlled ways. A programmer typically creates a mock object to test the behavior of some other object, in much the same way that a car designer uses a crash test dummy to simulate the dynamic behavior of a human in vehicle impacts.

6.2.2 Equivalence testing

This black-box technique minimizes the number of test cases. The possible inputs are partitioned in equivalence classes, and a test case is selected as a representative for each class. The assumption behind equivalence testing is that the system behaves in similar ways for all members of a class.

6.2.3 Boundary testing

This special case of equivalence testing focuses on the conditions at the boundary of the equivalence classes. The assumption behind boundary testing is that developers often overlook special cases at the boundary of the equivalence classes.

A disadvantage of these techniques is that they do not explore combinations of input data.

6.2.4 Path testing

This white box technique identifies faults in the implementation of a component. The assumption behind path testing is that, by exercising all possible paths through the code at least once, most fault will trigger failures. The identification of paths requires knowledge of the source code and data structures.

- *Statement Coverage criteria*: all lines of code are executed.
- *Decision Coverage criteria*: all branch statements must be executed (both conditions).
- *Condition Coverage criteria*: all conditions (in branch statements, loops,...) must be executed.

Cyclomatic complexity It can be shown that the minimum number of tests necessary to cover all edges (branches) is equal to the number of independent paths through the flow graph of a particular program. This is defined as the *cyclomatic complexity* (CC) of a graph.

$CC = |E| - |V| + 2P$ where $|E|$ is the number of edges of the graph, $|V|$ is the number of nodes of the graph and P is the number of connected components.

6.2.5 State-based testing

State-based testing focuses on object-oriented systems and compares the resulting state of the system with the expected state. In the context of a class, state-based testing consists of deriving test cases from the UML state machine diagram for the class.

6.3 Integration testing

Integration testing detects faults that have been not been detected during unit testing by focusing on small groups s of components. Two or more components are integrated and tested, and when no new faults are revealed, additional components are added to the group. This procedure allows the testing of increasingly more complex parts of the system while keeping the location of potential faults relatively small.

6.3.1 Horizontal integration testing strategies

Integrates components according to layers.

The *big bang* strategy assumes that all components are first tested individually and then tested together as a single system.

The *bottom-up testing strategy* first tests each component of the bottom layer individually, and then integrates them with components of the next layer up. A major advantage of this technique is that no stubs are needed.

The *top-down testing strategy* unit tests components of the top layer first, and then integrates the components of the next layer down. The advantage of this technique is that interface faults can be more easily found.

The *sandwich testing strategy* combines the previous two.

6.3.2 Vertical integration testing strategies

Integrates components according to functions. It focuses on early integration. For a given use case, the needed parts of each component, such the user interface, business logic, middleware and storage, are identified and developed in parallel and integration tested. This technique is used very often by agile methodologies.

6.3.3 Regression testing

Object-oriented development is an iterative process. When modifying a component, developers design new unit tests exercising the new feature under consideration. However, they should not assume that the rest of the system will work with the modified component, since the modification could introduce side effects or reveal previously hidden faults in other components. Integration tests that are rerun on the system to detect such failures are called *regression tests*.

6.4 System testing

Once the components have been integrated, system testing ensures that the complete system complies with the functional and non functional requirements.

- *Functional testing* finds differences between the functional requirements and the system. It is a black-box technique since test cases are derived from the use case model.
- *Smoke testing* consists of preliminary testing to reveal simple failures severe enough to reject a prospective software release. A subset of test cases that cover the most important functionality of a component or system is selected and run, to ascertain if the most crucial functions of a program work correctly.
- *Performance testing* finds differences between the design goals selected during system design and the system. Because the design goals are derived from he nonfunctional requirements, the test cases can be derived from the RAD. The following tests are performed: stress testing, volume testing, security testing, timing testing, recovery testing.

- During *pilot testing* the system is installed and used by a selected group of users. Users exercise the system and if it had been permanently installed.
- In *acceptance testing* the customer performs usability, functional and performance tests in the target environment. After acceptance testing, the client reports to the project manager which requirements are not satisfied.