

## PROJECT MANAGEMENT

### Centralised & distributed RCS

- **Revision control:** process of managing multiple ver of a piece of info (+: track history and evolution of proj, easier to collab, recover from mistakes, work simul on multiple ver of proj).
- 1. **Forking workflows** -> forks + PRs (+: no need write perms to main repo, only those merging PRs. - : extra overhead from forking)
- 2. **Feature branch flow** -> push/pull from same repo (protect main branch to reduce risk as all hv write perms)
- 3. **Centralised flow** -> everything done in master

### Software development life-cycle (SDLC)

- Stages: reqs, analysis, design, implementation, testing
- **Sequential:** each stage produces artifacts used in next stage. Only moves in forward direction. (waterfall) [linear process]
- **Iterative:** each iteration produces a new ver of product.
- Projects can be a mixture of seq and tier.
- **Agile:** rough proj plan that evolves w time (XP, Scrum)

### Work breakdown structure (WBS)

- Info about tasks and details in terms of subtasks.
- **Milestone:** end of a stage w significant progress.
- **Buffer:** time set aside to absorb delays (DO NOT inflate task est to create hidden buffers -> harder to detect incorrect effort est)

IDE - Integrated development environment

## REQUIREMENTS

- Stakeholders: users, customers, devs, dev company, sponsors, int grps, govt agencies, public, etc.
- Prototypes can uncover (how users interact w the system), discover, specify reqs.

**User stories:** can omit {benefit} if obv. Can add characteristics to {user role} (eg. forgetful)

- **Epics:** high-lvl user stories, cover bigger functionality, break down into multiple normal US.
- Add conditions of satisfaction to US for it to be 'done'. Others: priority, size, urgency.
- Capture user reqs for scoping, estimation and scheduling.
- Can also capture NFRs as US benefits stakeholders.
- Differ from trad reqs mainly in detail -> only enough to make an est of how long it will take to implement.

**Use cases:** captures functional reqs of a system

- Can involve multiple actors. Actors can be involved in many use cases. A single person/system can play multiple roles.
- Use cases can be specified at various lvls of detail.
- Describes only the externally visible behaviour, not internal details of a system (eg. saves file to cache)
- A step gives the intention of the actor, not the mechanics -> omit UI details -> give flexibility to UI designer
- Use case desc can show loops (eg. steps x repeated...)
- **Main success scenario (MSS):** desc the most basic interaction for a given use case, given nth wrong.
- **Extensions:** added to the MSS that desc alt flow of events.
- A UC can include another UC using underlined text (UC...) -> reduce clutter w low-lvl steps, repeated in multiple UCs.
- **UC diagrams:** actor generalisation using inheritance symbol (eg. Blogger can do all the UCs of Guest)
- +: simple and easy to unds, decouple user intention from mechanism by excl UI -> more freedom in how to provide functionality to user, encourages us to consider all situations that the software might face, separating typical and special cases encourages us to optimise typical cases.
- - : not good for capturing reps that do not involve a user interacting with the system -> also use not UCs for reqs

**Glossary:** domain-specific/technical terms, terms w multiple meanings, when multiple terms used for same concept.

### Architecture styles (cont.)

- **Event-driven:** controls flow of app by detecting events from event emitters and communicating those events to interested event consumers (eg. GUIs)
- **Svc-oriented:** builds apps by combining functionalities packaged as programmatically accessible svcs (eg. XML)

## DESIGN

**Coupling:** the measure of a degree of dependence btwn components, classes, methods, etc. Low coupling -> less dep.

- X is coupled to Y if a change to Y can potentially req a change in X.
- — of high coupling: harder to maintain, integration harder, testing and reuse harder.
- Eg. A is coupled to B if: (1) A has access to the internal structure of B (2) both depend on same global var (3) A calls B (4) A receives an obj of B as a param/return val (5) A inherits from B (6) both required to follow same data format or communication protocol.

**Cohesion:** a measure of how strongly-related and focused the various responsibilities of a component are. Higher cohesion keep related functionalities together while keeping out unrelated.

- — of low cohesion: lowers understandability, lowers maintainability (contain unrelated code), lowers reuseability.
- How: (1) code related to a single concept kept together. (2) code invoked close in time kept together (3) code that manipulates the same data structure kept together.

**Modelling:** to analyse a complex software dev entity, comm info to stakeholders, blueprint for creating software.

① **Class diagrams:** describe structure (NOT behaviour)

- Can omit 'methods' or 'attributes' box if not relevant.
- Underlines indicate class-level variables and methods.
- Association can be shown as an attribute instead of a line. Assoc roles are optional, good for showing multiple assoc between the same two classes.
- Attribute multiplicity and default values are optional.
- Common multiplcty: (1) 0..1 -> optional (2) 1 -> compulsory (3) \* -> 0 or more (4) n..m -> n to m inclusive.
- Inheritance: does not matter if the arrow is filled/empty
- Only indicate dependencies if it is not already captured by the diagram in another way like assoc or inheritance (eg. class A accessing a constant in B).

② **Object diagrams**

- Can omit attributes box and OBJECT NAME if not relevant.

③ **Conceptual class diagrams/OO domain model:** a lighter class diagram that captures class structures in the problem domain.

- Do not contain solution-specific classes (classes used in the sltn domain but do not exist in the problem domain). Eg. a DatabaseConnection class could appear in a class diagram but not usually in a CCD.

④ **Activity diagrams**

- Rectangles: ROUNDED corners, conditions: SQ brackets.
- Exactly ONE of the guard conditions should be true.
- Acceptable: omitting merge node, multiple arrows starting from same corner of branch node, omitting [Else] condition.
- Rake notation: indicates that it is in a separate diagram.

⑤ **Sequence diagrams:** model interactions between various entities in a system, in a specific scenario -> useful to verify the design of the internal interactions gives expected outcomes.

- Lifeline: when instance is alive
- Activation bar: when the method is being executed. Can't be too long or broken.
- Can omit ptional elems (eg. actvtn bars, return arrows). Informal operation descs can be used if precision not req.
- Can omit method params (eg. foo(...))
- Opt = optional path, par = parallel path, <<class>> = static methods to the class itself, ref/sd = in separate diagrams.

⑥ **Architecture diagrams**

- No standard notation but: minimise variety of symbols, explain uncommon symbols, avoid indiscriminate double-headed arrows to show interactions btwn components.

### Architecture styles

- Most apps use a mix of architectural styles.
- **N-tier:** higher layers make use of svcs by lower layers. Lower indep of higher layers (eg. OS and network comms)
- **Client-server:** server + client accessing svcs of server (distributed apps eg. online games and web apps)
- **Transaction-processing:** divides workload down of a syst down to a no of transacs which are given to a dispatcher that controls the execution of each transac (eg. banking)

**Design pattern:** an elegant reusable sltn to a commonly recurring problem within a given context in software design.

- Format: context, problem, sltn, anti-patterns (commonly used incorrect sltns, optional), conseq (optional), etc.

① **Singleton:** some classes should not have >1 instance.

- Prob: normal class can be instantiated >1 time.
- Sltn: make constructor priv. Provide a public class-lvl method to access the single instance.
- +: Easy to apply, effective w minimal work, easy way to access the singleton obj from anywhere in the codebase.
- — : singleton acts as a global var that incr coupling, difficult to replace singletons with stubs (can't override static method) in tests, singletons carry data from one test to another even when need tests to be indep of the others.
- Apply when there is risk of creating >1 by mistake w conseq.

② **Facade:** access functionality deep in other components

- Prob: access should be allowed w/o exposing internal details.
- Sltn: facade class sits btwn the component internals and users of the component -> all access through facade class

③ **Command**

- Prob: some parts of the code need to exec commands w/o having to know each command type.
- Sltn: general Command obj that can be passed, stored, exec w/o knowing the type of command (polymorphism)

④ **Model view controller (MVC)**

- Prob: most apps support storage/retrieval of info, displaying to user, changing stored info based on ext inputs -> high coupling due to interlinked nature of these features.
- Sltn: decouple data, presentation, and control logic; separate them into Model, View, and Controller.
- View: displays data, interacts w user, pulls data from Model.
- Controller: detects UI events and follows up, changes Md/Vw.
- Model: stores and maintains data, updates View.

⑤ **Observer:** some objs want to know changes (observe) to other objs

- Prob: observed obj does not want to be coupled to objects observing it. *relies on polymorphism. JAVAFX built*
- Sltn: force communication through an interface known to both

## **Design approaches**

① **Multi-level design:** eg, software (UI, Logic...) -> UI (UIManager...)

- Can be done top-down, bottom-up, or a mix.
- **Top-down:** design high-lvl design first, then lower. Useful for big systems where high-lvl needs to be stable first.
- **Bottom-up:** design lower-lvl components first then put them tgt to create higher. X: bigger systems, OK: variation of existing

② **Agile design:** not defined upfront, design emerges over time. Some initial architectural modeling at the very beginning.

## **Principles**

**Separation of concerns:** improves modularity, higher cohesion and lower coupling

- Reduce functional overlap, limits ripple effect during changes
- Can be applied at class lvl, as well as higher lvls
- Used by N-tier archi: each layer well-defined, no overlaps

**Law of Demeter:** objs should have limited knowledge of other objs

- Eg. void foo(Bar b) {Goo g = b.getGoo(); g.violatesLoD;}

## **QUALITY ASSURANCE**

- **Code reviews** (systemic examination, eg. PR review, pair prog, formal inspections). +: detect functionality defects and coding standard violations, verify non-code artifacts and incmplt code, does not req test drivers/stubs. — : manual -> error prone.
- **Static analysis:** can be done by IDEs (eg. find unused vars), linters are a subset of static analysers to make code 'cleaner'
- **Formal verif:** +: prove absence of errors —: only proves compliance, not actual util, expensive -> used in critical sftwrs.

**Testing:** feed input to SUT -> observe and compare actual & expctd

- **Testability:** indication of how easy to test a SoftwareUnderTest
- **Regression testing:** —: expensive -> practical to automate
- **Dev testing:** early testing GOOD as delaying until done bad: larger search space, inter-rltd bugs, may cause major reworks, bugs might hide others, delivery might be delayed if many bug
- **Stubs:** simple, so bugs in dependencies can't affect the test.
- Scripted vs expl testing: need a mix, expl depends on testers exp, detect errors in a short time, but scripted more systemic

*test driver is the code that 'drives' the SUT*

## **IMPLEMENTATION**

- IDEs consist of a source code editor, compiler/interpreter, debugger, etc.
- Use debuggers instead of tracing through code/using print.

**Refactoring:** restructuring code w/o modifying its ext behaviour

- Refactoring  $\neq$  rewriting. refactoring done in small steps.
- Refactoring  $\neq$  bug fixing: does not affect ext behaviour.
- Secondary benefits: hidden bugs become easier to spot, sometimes improves performance bc easier to optimise.
- Identify refactoring opportunities by code smells (eg. long methods, large class, duplicate code)
- Must balance the costs and benefits depending on context (eg. refactoring might lead to regressions).

## **Documentation**

- Two forms of dev-to-dev documentation (1) dev-as-user -> need to document how components are used (eg. API docs) (2) dev-as-maintainer -> need to document how systems are designed, implemented and tested for maintainers.

## **Assertions**

- Use liberally -> low impact on performance, worth the safety
- DO NOT use them to do work as they can be disabled.
- Suitable for verifying assumptions about internal invariants, control-flow invariants, pre/postconds, and class invariants.

## **Defensive programming**

- Not always necessary -> code more complicated, slower
- Depends on: how critical is the system? will the code be used by people not the author? level of language support, overhead

**Integration:** combining parts of a software to form a whole

- **Late-onetime:** wait for all components to be finished and int near the end. Bad as causes many incompatibilities.
- **Early-frequent:** int early and evolve each part in parallel, in small steps, re-int frequently.
- **Big bang:** int all or too many changes at the same time. Bad as it will uncover too many problems at the same time -> bug fixing more complex than incremental.
- **Incremental:** int a few components at a time, gradually.
- **Continuous integration (a type of build automation):** int, building, testing automatically done after each code change.
- **Continuous deployment:** changes int continuously and deployed to end-users at the same time.

## **Reuse**

- Increase robustness of system while reducing manpower/time
- Costs: overkill -> increase size and degrading performance, may not be mature/stable -> change rapidly, risk of dying off, licenses restrict use/development of software, may have bugs or security vulnerabilities, malicious code may sneak in
- **APIs:** specifies the interface through which other programs can interact with a software component, a contract btwn the component and its clients.
- **Libraries:** a collection of general modular code
- **Framework:** a reusable implementation of a software providing generic functionality that can be selectively customised to produce a specific app.
- Some frameworks provide a complete implementation of a default behaviour which makes them immediately usable.
- Frameworks facilitate the adaptation and customization of some desired functionality.
- Frameworks vs libraries: (1) libraries meant to be used 'as is' while frameworks are meant to be customised/extended. (2) your code calls the library code while the framework code calls your code -> frameworks follow the Hollywood principle.
- **Platforms:** provide a runtime env for applications. Often bundled w various libraries, tools, and technologies in addition to a runtime en, which is a platforms defining feature.

- **Acceptance tstg after system tstg:** passing syst  $\neq$  accptn (eg. two envs differ or system might conform to syst spec but not user req spec).
- **Autom testing:** use drivers, test GUI is hard -> move as much logic out of GUI
- **Equi partitions:** +: avoid testing too many -> incr efficiency, ensure all partition tested -> incr effectiveness and find bug %
- For EPs, consider: target object, input params, global vars.