Lecture Notes

CS261 - Software Engineering

Plan-Driven methodologies

1. **Software process model** *specifies* what it should do, how it should be *organised* and *implemented*, aligns with customer requirements (*validation*) and *evolves* over time.

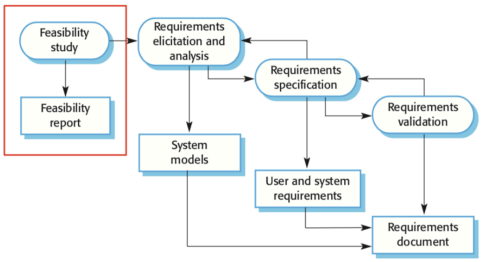
2. **Plan-driven**: all activities planned/fixed in advance, measure progress against initial plan. Few team constraints can outsource and independently test components (bc well planned). Easy to add members (churn). However, takes long time, hard to accommodate change or respond to updated customer requirements.

3. **Incremented Planning/development** delivers software in stages with customer feedback/acceptance testing. Cheaper to accommodate change, software is available to user quicker, better perceived value for money. But, hard to estimate cost, can lead to inconsistent design with evolving features, makes architectural changes harder over time, increases deployment overhead, and isn't cost-effective for documenting each version.

4. **Waterfall Model**: is plan driven, very rigid, sequential. Requirements analysis: system's services, constraints and goals, system design: software components and their relationships, Implementation and unit testing: test individual components, Integration and system testing: test system altogether, operation and maintenance: updates.

5. **Reuse-oriented Software Development**: Rewriting software is expensive, reuse common off the shelf (**COTS**) systems (frameworks). Requirements spec, component analysis: search for relevant components, requirements modification: analyse and modify them system design with reuse: design system with them, dev and integration: integrate them system validation.

6. Software spec written during **`requirements engineering`**



Find if task feasible, cost effective, derive requirements, put them in **system requirements document**, ensure they're achievable and valid, confirm with customer.

Agile methodologies

1. **Agile** planning is incremental, more adaptable to change, so much faster, but not well-documented, so harder to maintain.

2. Agile development supports customer involvement, incremental delivery, people not process: creativity is encouraged, embrace change, maintain simplicity.

3. Compared to plan-driven, agile has less emphasis on documentation and more on actual development.

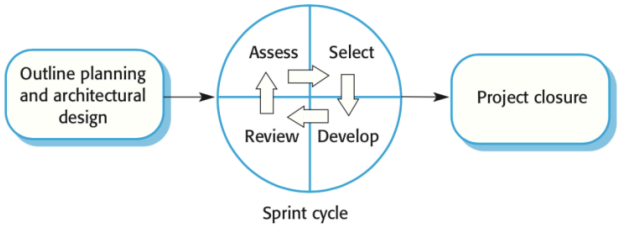
Area	Prototype	MVP
Purpose to	Test feasibility and proof of concept	Maximize validated learning for the least effort
Focus is	Presentation to stakeholders	Deployment into production
Features include	Some that may be discarded in the MVP	Basic and functional
Designed for	Small audiences – often stakeholders	Sizable customer groups
Legacy	Discarded after testing	Ist version of a complete solution
Feedback sought on	Product concept and idea	Product features and functionally
Composition may include	Mock-up, video, presentation	A functional basic product
Customer value	Demonstrates promised value	Delivers tangible initial value
Timing - Built when	Business case and product unproven, insufficient funds, risks unknown	Business case and product sound, sufficient funding, minimal risks
Testing	Market need	Product solution
Revenue	Not for sale	Sold to early adopters

4.

5. **Extreme Programming (XP)** has incremental delivery with fast iterations: several versions a day (*small releases)*, deliverables every 2 weeks, automated tests, continually refactor code, strong customer involvement, record requirements on story cards, at least 2 devs responsible for any part of code, integrate components as soon as ready.

   However, only works well in small experienced teams and requires heavy customer involvement.

6. **Scrum** prioritises iterative development: general goals: outline planning phase, sprint cycles: each cycle develops increment of system, project closure: wrap up.



Takes 2-4 weks, daily meetings, follow a to-do, features selected with customer, use when unstable requirements or shaky project.

Requirement Analysis

1. **Requirements** describe what system does, what service it provides and constraints on operation tailored to customer's needs; sets bases for tests, validation, verification and cost estimation.

2. Customer-facing **C-requirements** are targeting customer needs, and should be satisfied before the Developer-facing **D-requirements**. C-R presents to users, so simple natural language and diagrams. D-R targets developers, so exact project spec.

3. **Requirement-Set Qualities**: 1. *Prioritised*: time managed 2. *Consistent*: non-conflicting updates, 3. *Modifiable* and 4. *Traceable*: link to the source/reason.

4. **Individual Requirement Qualities**: Correct, Feasible, Necessary, Unambiguous and Verifiable.

5. Grade the requirements on Must/Should/Could/Won't to decide what to do.

6. Requirement analysis document must be understood by customers to ensure it meets their needs, managers to plan the system, engineers to implement, testers to design tests and maintainers to understand relationships between components.

7. Requirement analysis document comprises: Preface, Introduction, Glossary (not necessary), User requirements design, system architecture, system reqr spec, system models, system evolution, appendices.

8. **Functional requirements**: what system should do, which services to provide, how should read in scenarios.

9. **Non-functional requirements**: "qualities" like availability, performance, deployment, constraints on services.

10. Stakeholders have influence on system requirements, so need to deal with conflicting reqs, legal factors. Then need to **classify and organise** them, prioritise and negotiate, and finally formalise in requirements spec.

11. **Requirements Specification** comprises validity: does system support customer needs, consistency: are there conflicts, realism: is system feasible, verifiability: once complete, can system satisfy the requirements.

12. **Requirements Validation**: systematic manual analysis (review) involving both customer and developers, can prototype, use test-case generation for each requirement.

13. **Requirements management**: *problem analysis*: checking if reqs valid and unambiguous, *change analysis*: find effects of integration on the system. *Change implementation*: if valid, change requirement docs and implement.

Project Management

1. Main goals: meet deadlines, don't exceed budget, meet customer's expectations, maintain proper team wellbeing.

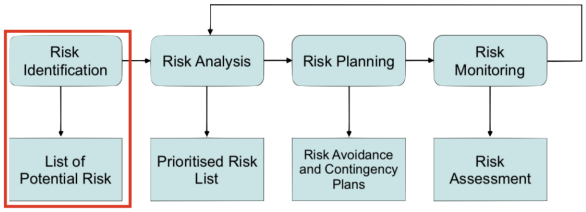
2. **People management**: **consistency**: make them feel valued, **respect**: give equal contribution opportunity, **inclusion**: consider everyone's views, **honesty**.

3. **Project Manager** is responsible for project planning, reporting, risk/project management. **Business analyst** introduces change to deliver value to stakeholders.

4. **Software Architect**: develops design of software given customer requirements. **Designer**: creative front-end developer, **Software Developer**: programmer, **Software Tester**: tests using black and white box techniques.

5. **Informal groups** work in competent teams, **hierarchical groups** have management levels, so better when system can be broken into subproblems.

6. **Risk management** identifies, assesses, prioritises risks.



7. **Project risks** affect schedule or resources (staff turnover, hardware malfunction, requirements change), **product risks** - quality or performance of software (tool underperformance) and **business risks** affect the organisation (technology change, competition).

8. **Risk types**: **technology** risk (soft/hardware), **people** risk (team), **organisational** risk (environment), **tool** risk (development software), **requirements** risk (adjust to change), **estimation** risk (approximate cost of resources).

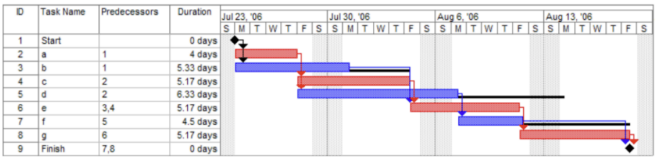
9. **Risk severity** can be catastrophic (threatens project survival), serious (delays), tolerable (still meet deadline), insignificant (ignore). Fix using **avoidance**: reduce probability of risk occurring, **minimisation**: reduce risk impact, **contingency plans**: have plan in case risk happens.

10. **Risk register**: identify risk, grade on a relative impact - relative likelihood table, find residual risk given a plan.

11. **Project planning** has **proposal phase**: plan resources, **startup phase**: role assignment and project breakdown, and **periodical planning**: adjust to changing reqs.

12. **Project Scheduling**: identify activities (tasks timeline) and dependencies (relationships between tasks), estimate resources needed, then allocate people .

13. **Gantt Chart**: show tasks against time.



14. **Schedule estimation**: **experienced-based**: use past experiences to estimate timelines, **algorithmic cost-based**: calculate the time given complexity and resources.

15. **Success measurement**: how it meets original specification (quantitative), customer's expectations (qualitative).

16. Problems with collaborating: code duplication, erasure, lack of accoutnability and introduction of errors, use Git.

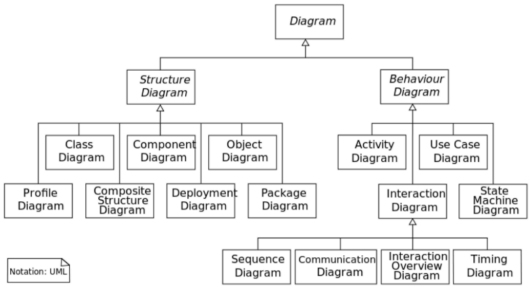
System Design

1. **Requirements analysis**: building contract between natural language/diagram based **customer** and unambiguous mathematical **developer**.

2. **System modelling**: developing quantitative models to clarify functionality, provide development basis and inform component-level decisions.

3. System modelling **perspectives**: **external**: model system context, **interaction**: interdependencies, **structural**: organisation, **behavioural**: dynamics.

4. **Unified Modelling Language (UML)**: unambiguously represents static/structural view of the system (objects, attrs) and dynamic/behavioural view (collaboration between objects).



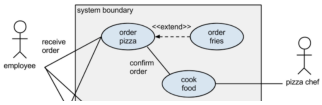
5. **Grammatical Approach**: natural language system description, **behavioural approach**: identify objects based on what participates in what behaviour, **scenario-based approach**.

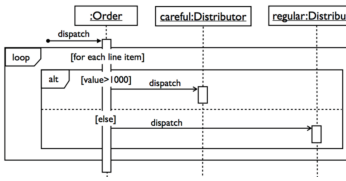
6. **Class Diagram**: UML notation showing system classes and their relationships. Don't include getters/setters or inherited methods, make all other methods public. Prefix interfaces with `<<interface>>` `(+) public, (-) private, (#) protected, (/) derived, (_) static, (~) package`.

7. Class **diagram association**: can connect with numbered lines to denote amount/**multiplicity**: (*) zero or more, (1) exactly one, $(x..y)$ inclusive range, $(x..*)$ x or more. Also has relationship **name** and **navigability**/direction. **Class hierarchy** arrows: class (solid, black), abstract class (solid, white), interface (dashed, white).

8. Associaiton can be one-to-one, one-to-many, "is part of" **aggregation** (white diamond), "is made up of" **composition** (black diamond), "uses temporarily" **dependency** (dotted line).

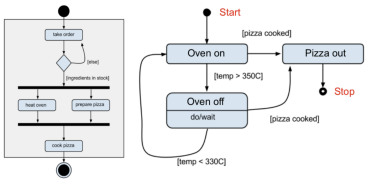
9. **Use case diagram**: represents user's (define interaction with a user class, not every singular user) interactions with the system (*left*).

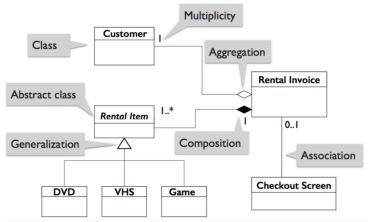




10. **Sequence diagram** shows temporal interaction (dashed line timeline) between processes, can terminate with a cross. Active processes are rectangles (*right*).

11. **Activity diagram**: action rectangle, condition diamond, parallelism line. **State diagram** shows how the system state changes.





12. **Structural/static Models** show organisation of components within the system. Formal methods include logic calculi, type stytems and algebraic data types, program semantics and automata theory.

Architectural Design

1. Architectural design is concerned with understanding how system should be organised. **Conceptual integrity**: conceptual vision of the solution, **quality driven** - embrace quality attributes, **recurring styles**: adopt architectures seen before, **separation of concerns**: reduce complexity.

2. Represented with box and line diagrams. Used to represent high-level system view for communication with customer, and model components and their relationship.

3. Targets non-functional requirements such as performance, security (levelled structure), safety, availability (fix issues

without stopping the system), maintainability since they target system as a whole.

4. **Pattern**: way to represent and share common knowledge.

5. **Layered Architecture**: separate system into independent layered modules s.t. they only rely on the level immediately below. Facilitates incremental design. However, layer going offline can affect entire system etc. Usually use for security and addons.

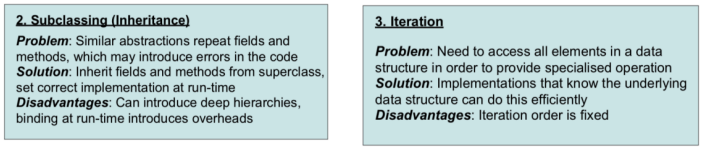
6. **Repository Architecture**: knowledge base, all interaction is done through it. Components can be independent, efficient data sharing, but have single point of failure, needs standardised data, so difficult to evolve. Useful for big data such as AI, aka "black board" approach.

7. **Pipe and Filter Architecture**: linear processing, focuses on runtime organisation, flexible, supports parallelism. Easy to understand and evolve, matches structure of many applications, but doesn't support GUIs and needs standardised data. Used in data processing applications.

8. **Model-View-Ctonroller (MVC) Architecture**: focus on how to interpret user interactions, update data and present that data to user. Useful for web-based systems. **Model** manages and updates data, **view** manages how it's present to users, **controller** manages user interactions and passes to view. Has separation of components, can easily change data, but overhead complexity, difficult to distribute development.

Creational Design patterns

1. **Design Pattern**: standard solution to a common programming problem. Comprises name, problem description, solution description and statement of consequences.



2. **Creational design patterns**: factories create objects, builders create complex objects (like step-by-step recipe), prototypes recreate by cloning.

3. **Factory** in OOP is a concise way to use constructors, but involves many classes which have to be recursively updated upon change.

4. **Builder**: creates complex objects comprising many different other objects. Different from a factory: simple constructors but requires many classes, not used as often.

5. **Prototype**: create new objects through cloning. Don't need another subclass to create it, keeps class hierarchy and complexity simple, but circular references might cause problems + have to update cloned object.

Structural Design patterns

1. **Structural design patterns** ease design and implementation by succinctly managing the relationships between objects.

2. **Proxy Pattern**: create placeholders for other objects (e.g. cover of a movie that when clicked redirects the user to the movie itself), "load on demand". Add more complexity through classes, but provides availability when not ready, manages object life cycle etc.

3. **Virtual proxy** (lazy initialisation): delay resource-heavy content loading with proxy. **Protection proxy**: access control requiring credentials. **Remove proxy**: offer client all functionality, but on another server. **Logging proxy**: keep track of access and requests to a server. **Caching proxy**: save results of an objects for later. **Smart referencing**: garbage collection (if nobody references heavyweight object - delete it).

4. **Decorator pattern**: adds new behaviour to objects at runtime. Inheritance is static. Can extend behaviour without adding new subclasses, but removing wrappers from stack is difficult, also order specific and

5. **Adaptor Pattern**: change data format dealt with to make it easier to manage without changing the underlying type (don't change mph speedometer, just translate into kph). However, wrappers make the code very complex, promotes single responsibility principle (one object handles conversion).

6. **Flyweight pattern**: get more objects in the memory (multiple object copies reference shared attributes). Saves memory but some data may need to be recalculated at each call, complicated code.

7. **Bridge pattern**: decouple abstraction from its implementation so that the two vary independently. **Composite pattern**: tree structure of same-interface objects.

8. **Facade Pattern**: create simplified interface of existing interface to ease usage in common tasks.

9. **Pipes and filters**: chain of processes where output of each process is input of the next (like monads).

Behavioural Design patterns

1. **Behavioural patterns** describe how objects communicate: either change their own internal data or interact by passing data to another object.

2. **Iterator Pattern**: traverse a container to access its elements without exposing the structure. Supports parallel runs. Extracts traversal logic out of the class making it simple (single responsibility principle), but may be redundant/ineffective.

3. **Observer Pattern**: automatically notify obj dependents on state change. Works for both push/pull models: producer/consumer, publish/subscribe. Publisher maintains list of subscribers who **opted in** for each event to avoid sending updates to non-subs. Relationships can change at runtime, but subscribers are notified in random order.

4. **Memento Pattern**: save and restore objects without revealing the details of is implementation (undo action). Originator object makes snapshot of itself and stores it in Memento within a Caretaker. Backups follow encapsulation and extract history maintenance keeping original classes simple, but takes a lot of memory, caretakers need to erase unneeded mementos, dynamic programming languages may modify states at runtime.

5. **Strategy Pattern**: select the method (strategy) to complete a task at runtime. Original class becomes a context object, deciding on which other patterns/classes to follow depending on context of the problem. Can swap implementations at runtime, simplifying the class hierarchy, but requires clients to understand key differences between strategies, anonymous functions are making this approach obsolete.

6. **Chain of Responsibility Pattern**: command objects are handled or passed on to other objects by logic-containing processing objects.

7. **Mediator Pattern**: provides a unified interface to a set of interfaces in a subsystem.

8. **Scheduled-task Pattern**: a task is scheduled to be performed at a particular interval or clock time.

9. **SOLID**
   • **Single responsibility**: a class should encapsulate the needed data and functionality for just one thing.
   • **Open** for extension/**closed** for modification **principle**: once designed and complete, extend the class, not edit.
   • **Liskov substitution principle**: behavioural subtyping, object that uses a parent class can use the other child classes without knowing.
   • **Interface segregation Principle**: improve maintainability by keeping things decoupled.
   • **Dependency inversion Principle**: interactions should rely on well defined interfaces and go from low level to high level to minimise num of dependencies.

Human Computer Interaction (HCI)

1. **Attention**: **Selective**: tune out things to focus on some stimuli, **Divided**: focus on multiple things at once, **Sustained**: attention span, **Executive**: keep track of steps and goals in sustained attention.

2. **Memory** has sensory stores: perceptions, working memory: transitory info, long-term. Memory decays overtime (decay theory), and new memory overwrites the old memory (displacement theory); proactive interference: can't retrieve memory because it's stored in the wrong place. Episodic memory: serial historical form, semantic: record of facts. Focus on short term memory! Icons might be resemblance: analogous image, exemplar, symbolic: high abstraction, arbitrary.

3. **Cognition**: process by which you gain knowledge: reasoning, understanding etc. **Norman's human action cycle**: form a goal, intention to act, planning to act, execution, feedback, interpret feedback, evaluate outcome. **Gulf of evaluation**: psychological gap which must be crossed to interpret a user interface - minimise load to understand the UI, **Gulf of execution**: gap between user's goals and the means to execute them (number of steps to complete an action).

4. **Gestalt principles**:
   • **Figure ground**: people tend to segment their vision into the figure (foreground) and the ground (background).
   • **Similarity**: if two things look similar people assume they behave similar.
   • **Proximity**: if two things are close by, people assume they must be related.
   • **Common region**: if two things are in the same box/region, people assume they are related.
   • **Continuity**: series of objects in a line or curve are perceived as related (Bezier curves are useful).
   • **Closure**: we fill in the blanks for complex patterns (seeing faces where there are none).
   • **Focal point**: you're drawn to the most unique/obvious piece of the image first.

5. **Affordances** are what an object allows us to do (opening/closing the door). Made clear with **signifiers** (labels), which can be perceptible (crossings) or invisible (the road is walkable anywhere). Can be false, but must always

be perceptible. Feedback: give user info on actions performed, constraint: retrain some interactions, mapping: relationship between controls and their effects, consistency: similar operations should use similar elements for similar tasks.

6. **Neilson's Usability principles**:
   • **Visibility of system status**
   • **Match between system and real world** (use user's langauge rather than system terms)
   • **User control and freedom**: provide escape routes (undo buttons)
   • **Consistency and standards**: avoid ambiguous terms.
   • **Help users recognise and recover from errors**: use natural language error descriptions with solutions.
   • **Error prevention**: prevent users from making mistakes wherever possible.
   • **Recognition rather than recall**: make all options visible rather than force the user to remember how to find them (Blender is terrible for this).
   • **Flexibility and efficiency of use**: provide accelerators for experts to do things faster (e.g. keyboard shortcuts)
   • **Aesthetic and minimalist design**: avoid using unimportant info, keep it simple.
   • **Help and documentation**: provide searchable information with concrete readable solutions.

7. Metrics for solution evaluation: Ratio of success to failure, Time to complete task, # of errors a user makes, # of times a user expresses frustration or satisfaction.

Dependability

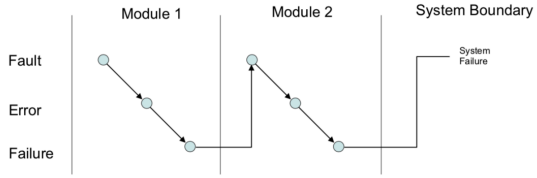
1. Attributes of **dependability**:
   1. Availability: likeliness of service being ready for use when invoked.
   2. Reliability: likeliness of the system providing designated service for a specified period of time. **Perceived** reliability: how reliable the system appears to its users.
   3. Safety: extend to which a system can operate without damaging or endangering its environment.
   4. Confidentiality: non-disclosure of undue information to unauthorised entities.
   5. Integrity: the capacity of computer system to avoid altering, withholding or deleting the information.
   6. Maintainability: function representing probability that failed computer system will be repaired in $\leq t$ time.

2. **Reliability measures**: *probability of failure on demand*: likelihood of system failing if someone makes a request for service. *Rate of occurrence of failures*: expected number of failures in a given time period. *Mean time to failure*: avg time of system running without failing. *Availability.*

3. **Relevant system properties**: **Repairability**: how easy to repair system when breaks. **Maintainability**: is it economical to add new requirements and keep the system relevant. **Error tolerance**: avoid user input errors.

4. **System failures**: **Hardware** fails because of design and manufacturing errors, or end of components' natural life. **Software** fails due to errors in specification, design or implementation. **Operational** or human mistakes.

5. **Fault-Error-Failure** cycle.



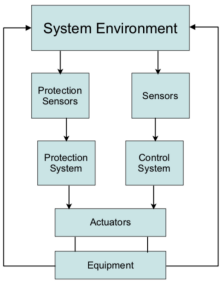
   Can avoid with fault **avoidance**, **detection and correction** (discover and remove faults before deployments) and fault **tolerance** (deal with mistakes).

6. Error **detection** and **recovery**:
   • **Graceful degradation**: enable system to operate in reduced capacity in the event of failure of some of its components.
   • **Redundancy**: include spare capacity in a system to be used if part of the system fails.
   • **Diversity**: enforce redundant components of different types to decrease probability of them failing in the same way.

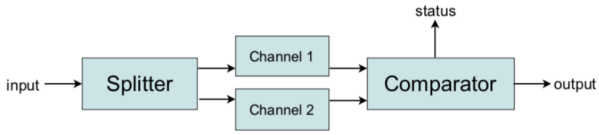
7. **Dependable processes**:
   • **Documentable**: have defined repeatable process model that sets out the documentation that must be created.
   • **Standardised**: applicable for many different systems and should contain a set of standards that apply to all.
   • **Auditable**: understandable by people other than those using them to enable verification (traceable static testing)
   • **Diverse**: include redundant and diverse verification and validation techniques (fault tolerance).
   • **Robust**: able to recover from failures of individual process activities (fault tolerance).

8. Dependability is all about the context, requires consideration of the entire system not just the intangible software.

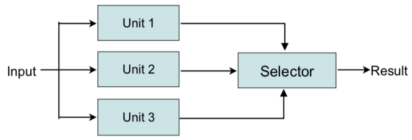
9. **Protection systems** monitor control system, equipment and environment, performs some action and moves system to safe state (shutdown) upon faults.



10. **Self-monitoring** architectures: monitor their own operation, operate on separate channels and compare outputs to ensure correct outputs.



11. **N-version programming**: multiple software units made by different teams under the same specification. Each version executed separately, outputs compared using a voting system, ignore inconsistencies, but usually costly and impractical.



12. Overall, rely on diversity: give problem to several separate teams who don't communicate and compare their outputs. The chance of all of them being wrong is low.

System Testing

1. Testing shows that program does what it was indented to do, highlights defects before the software is in use, forms part of verification and validation, demonstrates the software meets its requirements, but can only show presence of errors, not their absence.

2. **Static Testing** (without execution): code review, walkthroughs and inspections "does code meet the spec". Studies quality, compliance and maintainability, not only correctness. Can use support tools. Allows to consider code quality, hidden (interaction) errors and works with incomplete code. however bad at finding performance issues and unexpected interactions between components.

3. **Dynamic Testing**: executing code test cases to validate "does product meet needs of the customer". **Structural** (**white-box**) testing: control/data flow of system. **Functional** (**black-box**) testing: formal component specification, no view of the code.

4. **Statement adequacy**: all statements've been executed by ≥ 1 test. **Statement coverage** $= \frac{\#\text{ executed statements}}{\#\text{ statements}}$. Also have branch adquacy/discovery, condition coverage.

5. **Black-box** testing process:
   • Identify functions the software is expected to perform
   • Create input data and determine expected output based on function's spec.
   • Execute the test case and compare the actual and expected outputs.
   • Check if application meets customer's needs.

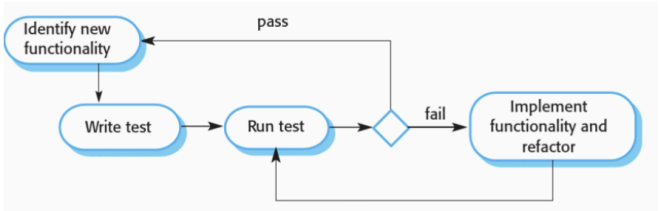
6. **Unit Testing**: initialise the system with inputs and expected outputs, then call the method and compare results. Set and check all attributes, put object in all possible states, simulate all events that cause a state change, can automate, but remember bout inherited classes that may violate assumptions, test them too.

7. **Integration Testing**: combine objects and methods, test their interactions. Unit tests may miss errors of this level.
   Interface Misuse: some component is not passing the right parameters or receiving an unexpected return value.
   Interface Misunderstanding: some interface doesn't understand behaviour of the other.
   Timing Errors: data producer may operate differently to consumer.

8. Check extremes of ranges, test interface calls with null pointer parameters, design tests that cause one of the interacting interfaces to fail and see how failure is handled. Stress test, especially in message passing systems. When components share memory, vary the order of components accessing it.

9. **System Testing**: check that components are compatible and interact as expected. may introduce off the shelf components, like integration testing for the whole system. **Emergent behaviour** are characteristics only seen when components interact, need to test for both expected and unexpected, usually with use-case testing and user interactions.

10. **Test Driven Development (TTD)**: develop tests for an increment of code, don't move onwards until all tests pass. Most effective when developing a new system, doesn't replace unit testing!



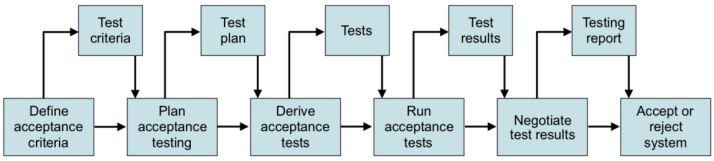
1. Identify a small, implementable, functional increment and write an ideally automated test.
2. Run this test before the increment, it **must fail** to verify it's working.
3. Implement the functionality (or refactor old code) and re-run the test, once all of them pass, move onto the next increment.

Other TDD benefits:
• **High code coverage**: each segment gets ≥ 1 associated test
• **Regression testing**: constantly verify that a new segment hasn't introduced bugs into the system.

• **Simplified debugging**: the issue must be in the newest segment of code.
• **System documentation**: tests double as documentation making it clear what the code is meant to be doing.

11. **Write the Test First** to clarify what the code segment is supposed to do, makes implementation simpler since there is a deep understanding. "if you don't know enough to write the tests, you can't develop the required code".

12. **User Testing** can be:
Alpha Testing: selected users work very closely with development team on very early versions of the software. Reduces risk of unanticipated changes to software disrupting business.
Beta Testing: much larger group of users experiment with more complete system version. Good for testing software with wide range of settings, discovers issues in interaction with the operating environment. Can server as marketing, let people learn about the software.
Acceptance Testing: customers test the system to decide if it's ready for deployment. Often used to determine if payment should be issued.



1. Define Acceptance Criteria: define what the user will determine as an acceptable system. Can be difficult, and may take chaning requirements into account.
2. Plan Acceptance Testing: establish resources, schedule and order for testing. Defines requirements coverage and mitigates poor performance.
3. Derive Acceptance Tests: test functional and non-functional aspects of the system, cover most/all requirements, make sure tests demonstrate meeting of criteria.
4. Write Acceptance Tests: define user journey as a series of steps, then define expected output, both aiming to cover the set of requirements.
5. Run Acceptance Tests: take place in deployment environment unless disruptive, difficult to automate, involves use of end-users who may need training on how to effectively test.
6. Negotiate Test Results: unlikely that all tests will pass, decide which problems are negligible and agree on how to fix any issues if possible.
7. Accept or Reject System: if unacceptable, decide how much more development to involve, otherwise deploy.

Release Management

1. **Version Control** allows the system to roll-back and manage external back-up (usually cloud). Especially useful in distributed code teams near-delivery, mitigates **human error**: user commits bad code last minute and **technical error**: laptop dies/gets stolen.

2. Can feasibly release **development**, **feature**, **master** and sometimes **test** software system environments. Version control allows to manage transition between these, and ensure you release the right one.

3. **System Building** is done near the end of the project. It links the system together, turning the codebase into a deployable program. For this integration can use:
• **Build Script Generation**: either automate this or edit scripts manually.
• **Version Control (VC) System Integration**: link the build system to VC system to ensure the right version for each component.
• **Minimal Recompilation**: system shouldn't recompile everything every time, only the needed components.
• **Executable System Creation**: ensure easy install/setup, link everything into a single executable.
• **Test Automation**: test the system before performing a build.
• **Reporting**: ensure system lets you know if the build succeeded.
• **Documentation Generation**: setup the build system to generate any needed documentation.

4. System building comprises **Development System**: different team members may have different versions, need to compile into a final version. **Build Server** needs to maintain the definitive version and link together all external sources. **Target Environment** includes databases and apps not present in development environment, consider these differences.

5. **Data Management**: dummy data/stubs differs from real/production data, so when transitioning to it, need to check: availability to data sources, constraints around data acquisition (standards, cost), overhead of data processing (memory, time), long-term storage/backups (cost, availability, capacity) and data sharing/access constraints. People are careful about the historical record of the data provenance **data** and its **origins**.

6. **Releasing Software** is **expensive**. Need to consider config, installation (installer), documentation, marketing, time of new version release (not too often or too rare).

7. Don't assume the user s on a particular version of the software or that it's unchanged.
   1. Dependencies should be explicit and recorded.
   2. Physical distribution should be hidden, not hardwired and via multiple channels.
   3. Release process should require minimal effort for both parties.
   4. Scope of release (to whom) should be controllable, useful to record downloads.
   5. Sufficient descriptive information should be available.
   6. Interdependent systems should be retrievable as a group.
   7. Unnecessary retrievals should be avoided.
   8. History should be kept (release log).

8. Factors to consider when implementing a change:
   • **Consequences of not implementing the change**: if the system is crashing, you must make the change, otherwise might not.
   • **Benefits of change**: is the change worth the cost.
   • **Number of users affected by the change**: is it useful to the public, will the users need to re-learn the system.
   • **Cost of making the change**: will it involve many components, how complex or time consuming etc.
   • **Product release cycle**: don't overwhelm users with releases, if change isn't urgent better wait.

9. Developers need to decide if the customer's suggestion should be implemented/given priority over other planned adjustments. Refactoring and quality improvement should also be left to the developers' discretion

Theme	**Keywords found in your notes**
Process models	Waterfall, Incremental, Prototyping, Spiral, XP, Scrum, Agile Manifesto, Backlog, Sprint, Retrospective
Requirements	Functional vs Non-functional, Elicitation, Validation, Stakeholders, MoSCoW, Use case, Traceability
Project management	Gantt Chart, Critical Path, Risk Matrix, Risk Exposure = P × L, Risk Mitigation, Scheduling, Motivation (Herzberg)
Modelling & Design	UML, Class diagram, Sequence diagram, State machine, Encapsulation, Cohesion, Coupling
Architectures	Layered, Repository, Client-Server, Pipe-and-Filter, MVC, Microservices, Architectural Style, Quality Attribute
Design Patterns	Singleton, Factory, Adapter, Composite, Observer, Strategy, Decorator, SOLID principles (SRP, OCP, LSP, ISP, DIP)
HCI	Affordance, Feedback, Consistency, Nielsen Heuristics, Visibility, Mental Model, Fitts's Law
Dependability & Safety	Fault, Error, Failure, Fault-tolerance, Redundancy, Reliability, Safety-critical, Hazard, MTTF, POFOD
Testing & Verification	Static analysis, Dynamic testing, Unit testing, Integration testing, TDD, UAT, Test coverage
Release & Maintenance	Version Control (VCS), Git, Branching, Build Automation, Semantic Versioning, CI/CD, Regression, Patch

Table 1: Buzzwords