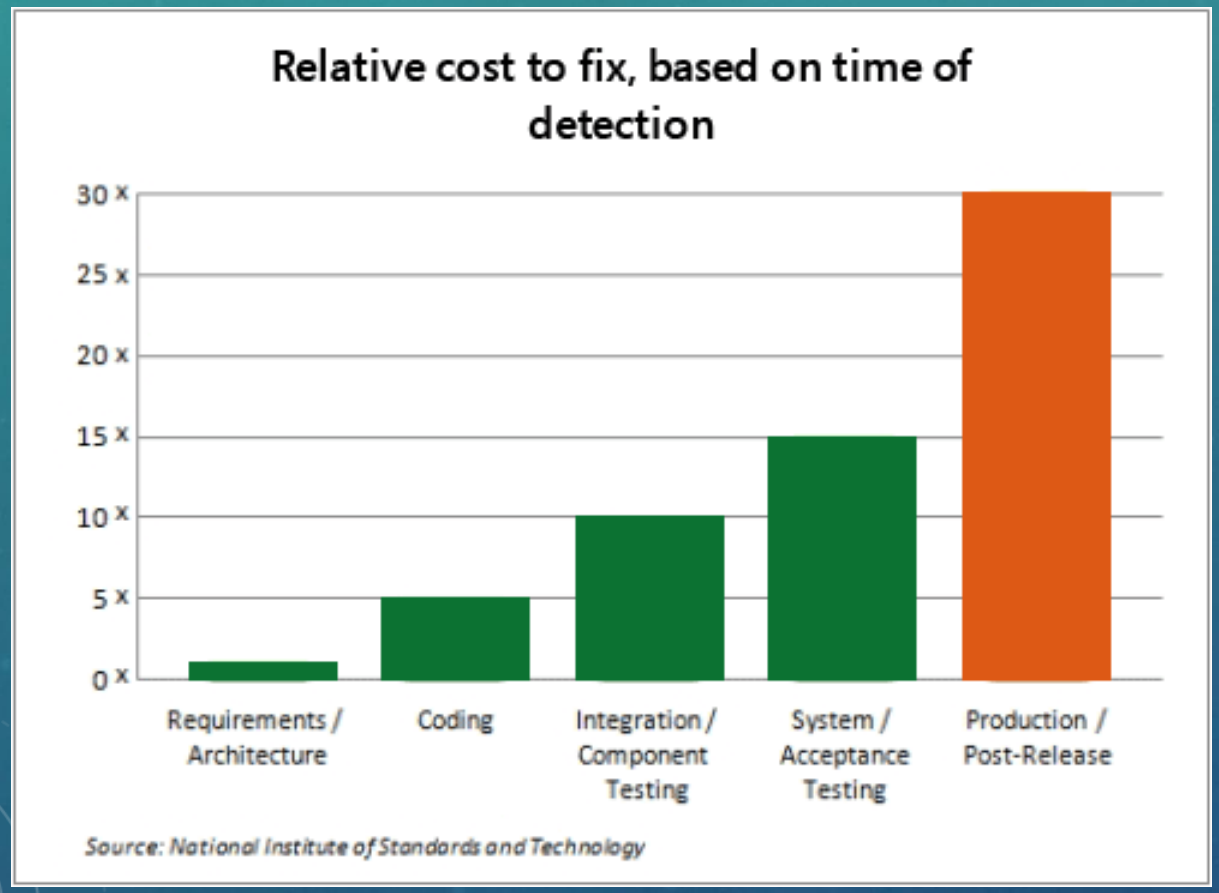
REQUIREMENTS VALIDATION

PART ONE: REQUIREMENT VALIDATION (CH17)

WHY VALIDATE REQUIREMENTS?

- It takes an average of 30 minutes to fix a defect discovered during the requirements phase of SDLC.
- In contrast, 5 to 17 hours will be needed to correct a defect identified during system testing.



WHY VALIDATE REQUIREMENTS?

- Requirements validation activities attempt to ensure
  - The software requirements accurately describe the intended system capabilities, functionality, appearance and properties that will satisfy the various stakeholders' needs.
  - The requirements must be
    - complete,
    - feasible, and
    - verifiable.

  (These are criteria of requirement validation.)

Verification determines whether the product of some development activity meets its requirements (doing the thing right). Validation assesses whether a product satisfies customer needs (doing the right thing).

HOW TO VALIDATE?

Informal peer review and formal peer review

- Informal peer review
  - A peer desk-check, in which you ask one colleague to look over your work product.
  - A pass-around, in which you pass on a deliverable to several colleagues to examine concurrently.
  - A walkthrough, during which you lead peers to read through a deliverable and note down comments the peers expressed.

HOW TO VALIDATE?

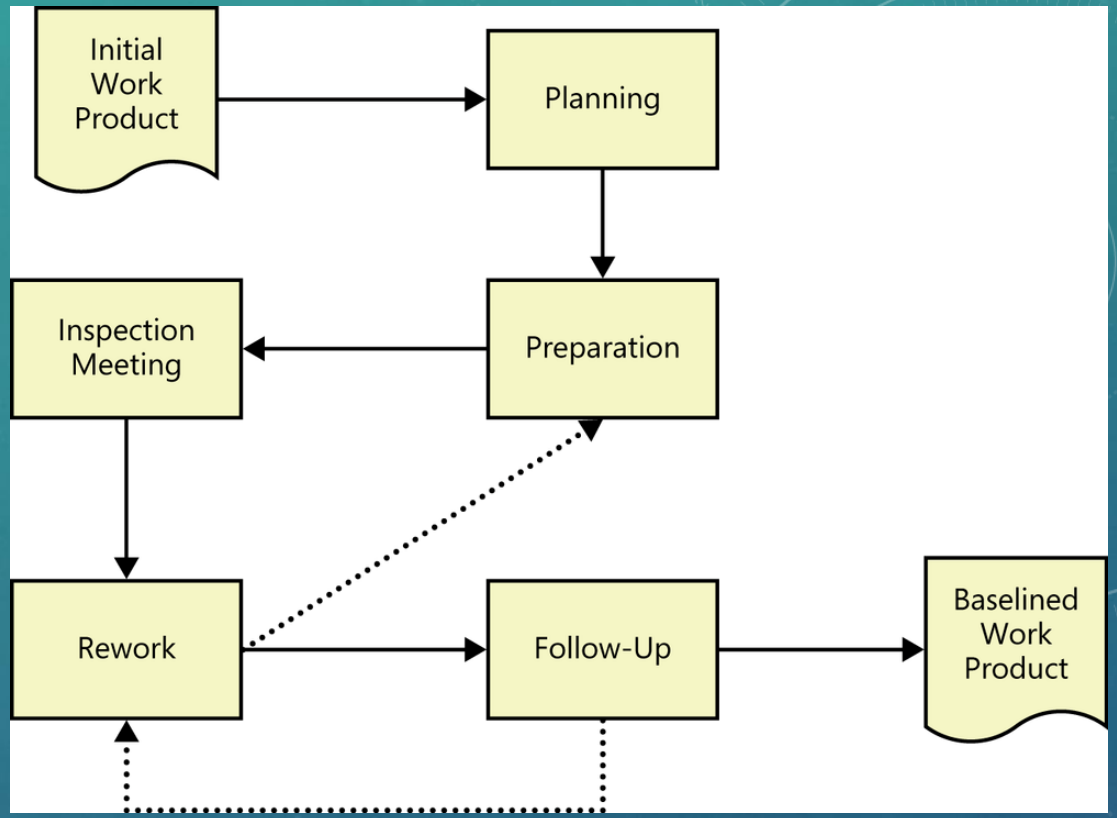
Informal peer review and **formal peer review**

- Formal peer review is a well-defined process called inspection process
  - Who will participate?
  - What are their roles?
  - What level the requirement documents should reach to start the review (entre criteria)?
  - What is the process?
  - When can the process end (exit criteria)?

HOW TO VALIDATE?

- Participants of formal peer review :
  - The author (BA) of the SRS and perhaps peers of the author
  - The actual user representatives who are the sources of requirements to be inspected.
  - The developers, testers, project manager, etc. who will do work based on the requirements to be inspected.
  - People who are responsible for interfacing systems that has related to the requirements to be inspected. They will look for problems with the external interface requirements.
- Roles in formal peer review :
  - Author -- creates or maintains the work product (SRS) being inspected
  - Moderator/chair -- plans the inspection with the author, coordinates the activities, and facilitates the inspection meeting
  - Reader -- One inspector is assigned the role of reader to represent SRS.
  - Recorder -- uses standard forms to document the issues raised and the defects found during the meeting.
  - Other inspectors – raise questions about item of SRS

HOW TO VALIDATE?



- Process of formal peer review
  - Planning: BA and moderator determine who should participate, what materials the inspectors should receive prior to the inspection meeting, the total meeting time needed to cover the material, and when the inspection should be scheduled.
  - Preparation: Prior to the inspection meeting, BA should share background information with inspectors so they understand the context of the items being inspected and the BA's objectives for the inspection.
  - Inspection meeting: During an inspection meeting, the reader leads the other inspectors through the document, describing one requirement at a time in his own words. As inspectors bring up possible defects and other issues, the recorder captures them in the action item list for the requirements author.
  - Rework: The BA should plan to spend some time reworking the requirements following the inspection meeting
  - Follow-up: In this final inspection step, the moderator or a designated individual works with the BA to ensure that all open issues were resolved and that errors were corrected properly.

HOW TO VALIDATE?

- Entry criteria for formal peer review – if SRS does not satisfy the conditions, peer review meeting should not be held:
  - The document conforms to the standard template and doesn't have obvious spelling, grammatical, or formatting issues.
  - Line numbers or other unique identifiers are printed on the document to facilitate referring to specific locations.
  - All open issues are marked as TBD (to be determined) or accessible in an issue-tracking tool.
  - The moderator did not find more than three major defects in a ten-minute examination of a representative sample of the document.
- Exit conditions – review process can be concluded when these conditions are satisfied:
  - All issues raised during the inspection (newly raised one in This meeting) have been addressed.
  - Any changes made in the requirements and related work products were made correctly. (This implies that the BA needs to change SRS according to inspectors' comments in the meeting and after.)
  - All open issues have been resolved, or each open issue's resolution process, target date, and owner have been documented. (This implies the next round review meeting is needed.)

HOW TO VALIDATE?

- Requirements review is challenging
  - Large requirement document -- might carefully examine the first part and a few hard-working reviewers will study the middle, but it is unlikely that anyone will look at the last part.
  - Large inspection team -- increase the cost of the review, make it hard to schedule meetings, and have difficulty reaching agreement on issues.
  - Geographically separated reviewers (particularly, for example, travel restrictions due to Covid-19)
  - Unprepared reviewers -- risk people spending the meeting time doing all of their thinking on the spot and likely missing many important issues

V-MODEL (REFERRING TESTS THAT DOSN'T TAKE PLACE AT REQUIREMENT STAGE

Software development life cycle

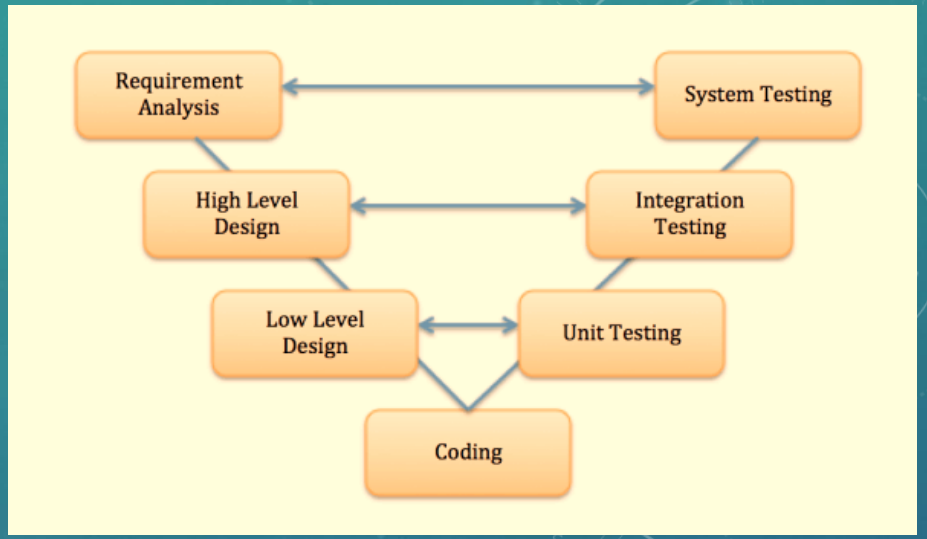
Software Development Life Cycle (SDLC)	Activities performed in each stage
Requirement Gathering stage	Gather as much information as possible about the details & specifications of the desired software from the client.
Design Stage	Plan the programming language like Java, PHP, .net; database like Oracle, MySQL, etc. which would be suited for the project, also some high-level functions & architecture.
Build Stage	After the design stage, it is build stage, that is nothing but actually code the software
Test Stage	Next, you test the software to verify that it is built as per the specifications are given by the client.
Deployment stage	Deploy the application in the respective environment
Maintenance stage	Once your system is ready to use, you may require to change the code later on as per customer request

V-MODEL

Concept of V-model (V – validation)

Testing is designed for every phase in SDLC.

- The left side of the model is Software Development Life Cycle - SDLC
- The right side of the model is Software Test Life Cycle - STLC
- V-model indicates
  - acceptance tests are derived from the user requirements (for low-level design – mistakes in design can be spotted earlier),
  - system tests are based on the functional requirements (new for requirement analysis – make sure key details not missing),
  - integration tests are based on the system's architecture (for high-level design – mistakes in architecture can be found earlier).



- Unit testing is O as we always test code and test environment when programming
- Integration test is OK as we have, for example, prototyping in traditional SE
- System testing at requirement phase
- How to test? What would be the testbed to perform the test? -- Refer to Slide 4 – complete, feasible and verifiable.

V-MODEL

	Throwaway	Evolutionary
Mock-up	Clarify and refine user and functional requirements. Identify missing functionality. Explore user interface approaches.	Implement core user requirements. Implement additional user requirements based on priority. Implement and refine websites. Adapt system to rapidly changing business needs.
Proof of concept	Demonstrate technical feasibility. Evaluate performance. Acquire knowledge to improve estimates for construction.	Implement and grow core multi-tier functionality and communication layers. Implement and optimise core algorithms. Test and tune performance.

Prototyping requirements for testing

- Provide peers with a model of the system to work with (??)
- All prototyping techniques can be used.

  (I think the key is to design testing scenarios.)

Testing requirements using the prototypes

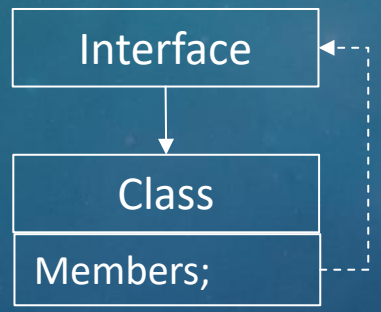
(Validation with respect to feasibility)

- Conceptual tests from user requirements covering the normal flow of each use case, alternative flows, and the exceptions you identified during elicitation and analysis
- Conceptual tests are independent of implementation
- Acceptance tests -- Agile development approaches often create acceptance tests instead of writing precise functional requirements. Each test describes how a user story should function in the executable software. Because they are largely replacing detailed requirements, the acceptance tests on an agile project should cover all success and failure scenarios.

AGILE SOFTWARE DEVELOPMENT – TESTING

Test-driven development (TDD)

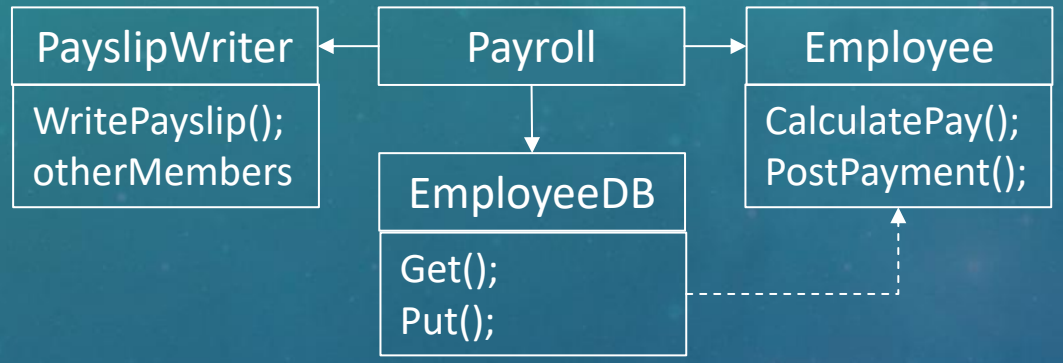
- Being able to design a test environment before actually developing/coding, showing developers' very clear thoughts about what the program is going to achieve and how it will operate.
- The obvious benefit of having such a test environment is that all functions of the program can be verified while they are developed.
- A test environment is actually an interface allowing developers to access the programs to be tested, either verifying or validating. (interface between programmers and functions/behaviours, but not interface class)



AGILE SOFTWARE DEVELOPMENT – TESTING

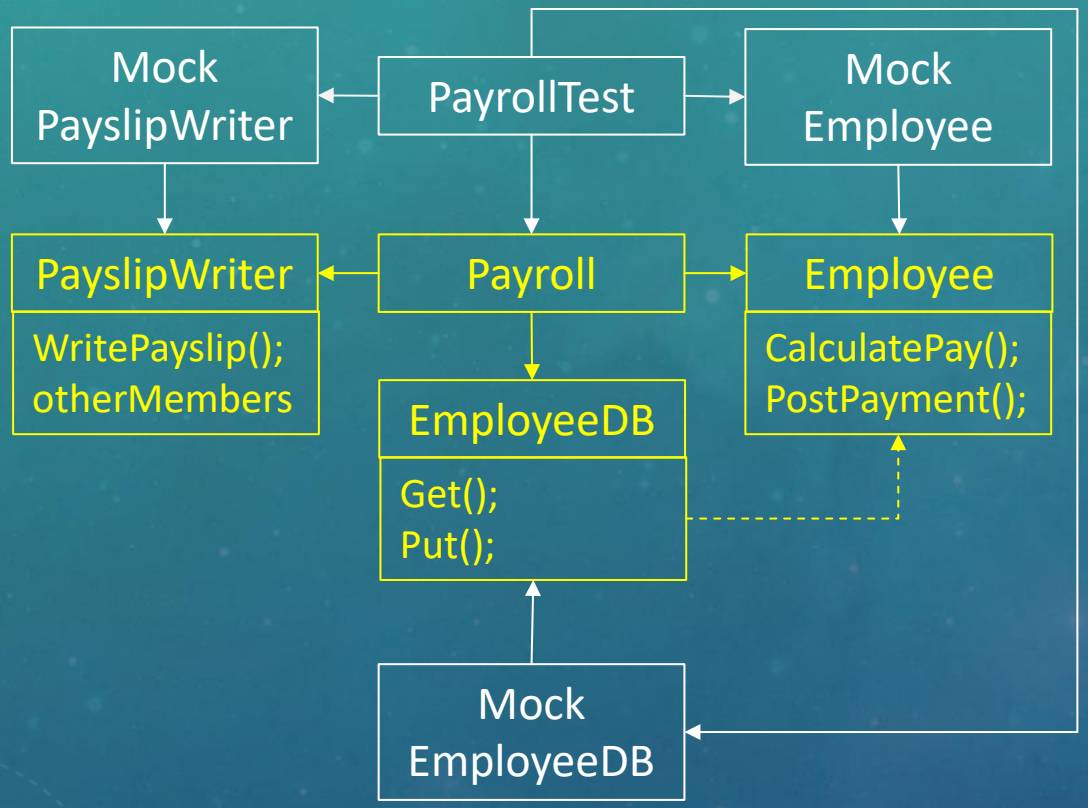
Example: suppose we want to create a payroll system which consists of Payroll class, Employee class, PayslipWriter class and Employee Database.

1. Payroll first calls EmployeeDB.Get() to have an employee's information
2. It then calls Employee.CalculatePay() to have the amount to pay
3. It then asks PayslipWriter.writePayslip() to print out the payslip for the employee



AGILE SOFTWARE DEVELOPMENT – TESTING

The test environment should look like this



- The “white line” system is the test environment and the yellowed ones are the program.
- MockEmployee, MockEmployeeDB and MockPayslipWriter are interfaces between PayrollTest and the corresponding “real” classes of Employee, EmployeeDB and PayslipWriter, respectively.
- It is a “prototype”.
- When Employee.CalculatePay() is developed, to test the method, PayrollTest calls the method through MockEmployee as if the latter has the method. MockEmployee then calls the method that stays in Employee class.
- The other methods are tested in the same manner when they are developed.

AGILE SOFTWARE DEVELOPMENT – TESTING

Acceptance test

- Tests at task and unit level versify the code, in the manner of “white-box test”.
- Acceptance test validates the implementation of an entire US in the way of “black-box test”, which means that the developers give inputs to the program and check whether the outputs are the ones they are expected.
- Acceptance tests, ideally, are designed before programming.
- Example: To test if the Payroll program is able to add and to retrieve employees’ records, we can input the following into the program via GUI (Payroll class)

  123445678, Dayou, Li, male, 2000

  and then search for the record and display it on the screen, also via GUI.

HOW TO VALIDATE?

Designing tests for validating requirements with acceptance criteria

Validation against acceptance criteria can be operated in the following manners:

- Starting from the high-priority functional requirements, i.e. those that must be satisfied so that the product could be accepted and used, or those that may not working quite right and must be fixed without delaying an initial release.
- Then moving to the essential non-functional criteria that must be satisfied based on the prioritisation of internal and external quality attributes.
- The next are the remaining open issues and defects that are left in the last round peer review meeting.
- In conjunction with specific legal, regulatory, or contractual conditions.
- Continuing with supporting transition, infrastructure, or other project (not product) requirements.

PART TWO: REQUIREMENTS REUSE (CH18)

ARE YOU CRAZE? IS IT POSSIBLE?

It is possible, see this example

- British Airways' website allows passengers to check in, to choose seats, to pay for seat upgrades, and to print boarding pass. Their self-service check-in kiosks at airports have the same functions. Their App also allow online check in, to choose seats and to pay for seat upgrades.
- Air France also has its website, App and airport kiosks, which have the same functions.
- You see the the same in requirements hence they are reusable across different products and different customers.

Benefits

- From developer perspective -- faster delivery, lower development costs, consistency both within and across applications, higher team productivity, fewer defects, and reduced rework.
- From user -- functional consistency across related members of a product line or among a set of business applications (Why this matters – think about website, app and kiosks of the same airline).

A 3D SPACE -- DIMENSIONS OF REQUIREMENTS REUSE

- Extent of reuse (X-axis, discrete) , values are
  - Individual requirement statement (a passenger wants to check in to a flight….)
  - Requirements plus its attributes (…. with seat confirmed and boarding pass QR code sent to mobile phone, …..)
  - Requirement plus its attributes, context, and associated information such as data definitions, glossary definitions, acceptance tests, assumptions, constraints, and business rules (…. to allow passenger to scan the pass at airport.)
  - A set of related requirements
  - A set of requirements and their associated design elements (such as class diagrams, interaction diagrams, etc.)
  - A set of requirements and their associated design, code, and test elements (such as test environment)

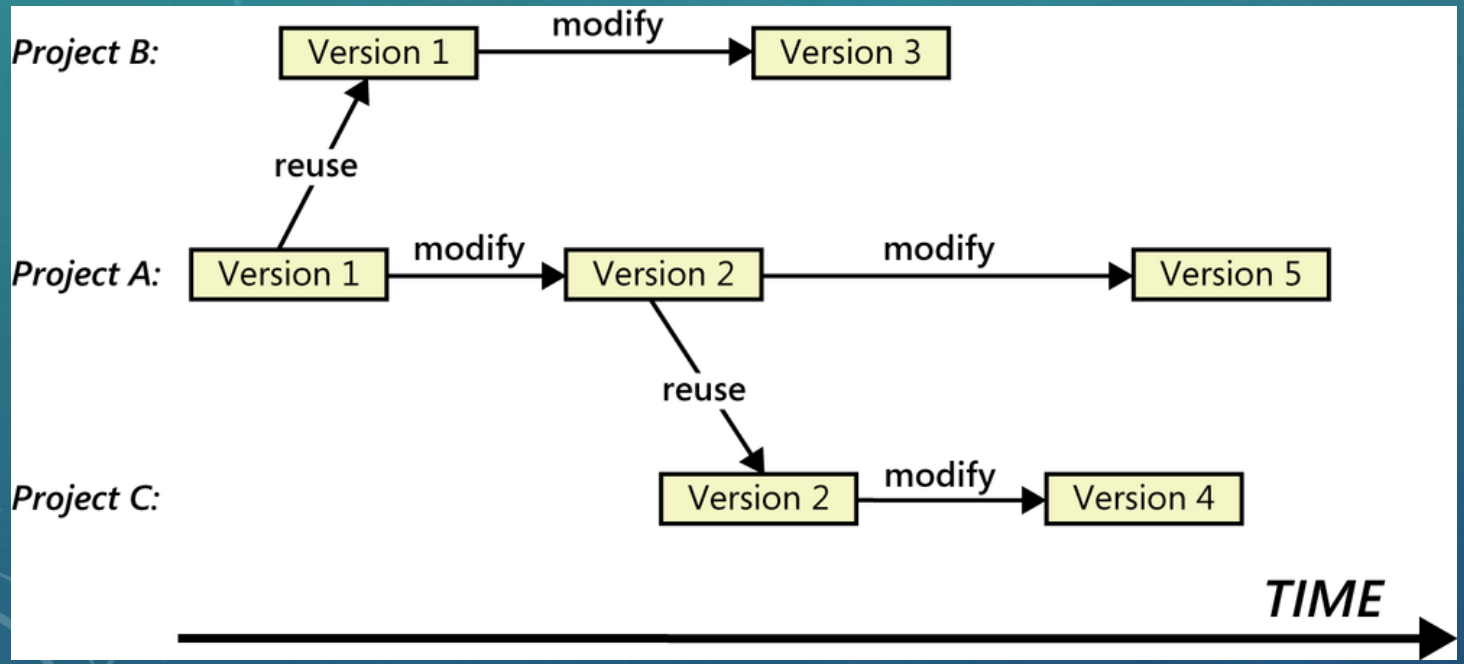
A 3D SPACE -- DIMENSIONS OF REQUIREMENTS REUSE

- Extent of modification (Y-axis, discrete as well) , values are
  - None, e.g. VSLAM required in aircraft inspection and in robot self-motivated learning
  - Associated requirement attributes (priority, rationale, origin, and so on), e.g. “high-quality images” has higher priority than “location information” in aircraft inspection, and the way around in robot leaning
  - Requirement statement itself (quantitatively but not qualitatively)
  - Related information (tests, design constraints, data definitions, and so on), e.g. drone flight path constraints in indoor inspection differs from that in outdoor

- Reuse mechanism (Z-axis) , values are
  - Copy-and-paste from another specification or another project
  - Copy from a library of reusable requirements
  - Refer to an original source

A GOOD PRACTICE

Having a DB storing different versions of requirements. If someone modifies that requirement in the database, the older version that you are reusing still exists. You can then tailor your own version of requirement to suit the needs of your project without disrupting other re-users (Concept of class override? Why not have a high-level (abstract or interface) requirement and then specify it for individual applications? – refer to “requirement pattern”.)



LIKELY REUSE OPPORTUNITIES

Some types of requirements-related assets that have good reuse potential

Scope of reuse	Potentially reusable requirements assets
Within a product or application	User requirements, specific functional requirements within use cases, performance requirements, usability requirements, business rules
Across a product line	Business objectives, business rules, business process models, context diagrams, ecosystem maps, user requirements, core product features, stakeholder profiles, user class descriptions, user personas, usability requirements, security requirements, compliance requirements, certification requirements, data models and definitions, acceptance tests, glossary
Across an enterprise	Business rules, stakeholder profiles, user class descriptions, user personas, glossary, security requirements
Across a business domain	Business process models, product features, user requirements, user class descriptions, user personas, acceptance tests, glossary, data models and definitions, business rules, security requirements, compliance requirements
Within an operating environment or platform	Constraints, interfaces, infrastructures of functionality needed to support certain types of requirements (such as a report generator)

LIKELY REUSE OPPORTUNITIES

Examples

- Within the same product – in Luton DPS CRM project, it is the case that info desk, consultant and finance all these three different users have the requirement of accessing the CRM system, though for different purposes.
- Across product line – Jingdong's web and app share the same business objectives and rules, as well as many user requirements.
- Across enterprise – British Airways, Air China, etc. share business rules, stakeholder profiles, user class descriptions and many others in their self-check-in kiosk systems
- Across business domain – Amazon and Natwest bank perhaps have the same security requirements.
- Within an operating environment or platform – Linux changed its GUI the same as Windows GUI.

Exercise: Imagine you are a BA in the project of developing Jingdong's App. Can you list 5 functional requirements that the App can share directly from Jingdong's website application and at least 2 functional requirements that you think need to modify?

LIKELY REUSE OPPORTUNITIES

Followings are some other groups of requirement information related to reuse

- Functionality plus associated exceptions and acceptance tests
- Data objects and their associated attributes and validations
- Compliance-related business rules, other regulatory constraints by industry, and organization policy-focused directives
- Symmetrical user functions such as undo/redo (if you reuse the requirements for an application's undo function, also reuse the corresponding redo requirements)
- Data objects and their related operations, such as create, read, update, and delete

LIKELY REUSE OPPORTUNITIES

Reuse opportunities

Reuse opportunities	Example
Business processes	Often business processes are common across organizations and need to be commonly supported by software. Many institutions maintain a set of business process descriptions that are reused across IT projects.
Distributed deployments	Often the same system is deployed multiple times with slight variations. This is fairly typical for retail stores and warehouses. A common set of requirements is reused for each separate deployment.
Interfaces and integration	There is often a need to reuse requirements for interfaces and integration purposes. For example, in hospitals, most ancillary systems need interfaces to and from the admissions, discharge, and transfer system. This also applies to financial interfaces to an enterprise resource planning system.
Security	User authentication and security requirements are often the same across systems. For example, the systems might have a common requirement that all products must have a single sign-on using Active Directory for user authentication.
Common features	Business applications often contain common functionality for which requirements—and perhaps even full implementations - can be reused. Possibilities include search operations, printing, file operations, user profiles, undo/redo, and text formatting.
Similar products for multiple platforms	The same core set of requirements is used even though there might be some detailed requirement and/or user interface design differences based on the platform. Examples include applications that run on both Mac and Windows or on both iOS and Android.
Standards, regulations, legal compliance	Many organizations have developed a set of standards, often based on regulations, that are defined as a set of requirements. These are reused between projects. Examples are ADA Standards for Accessible Design and HIPAA privacy rules for healthcare companies.

STANDARDISATION: REQUIREMENT PATTERNS

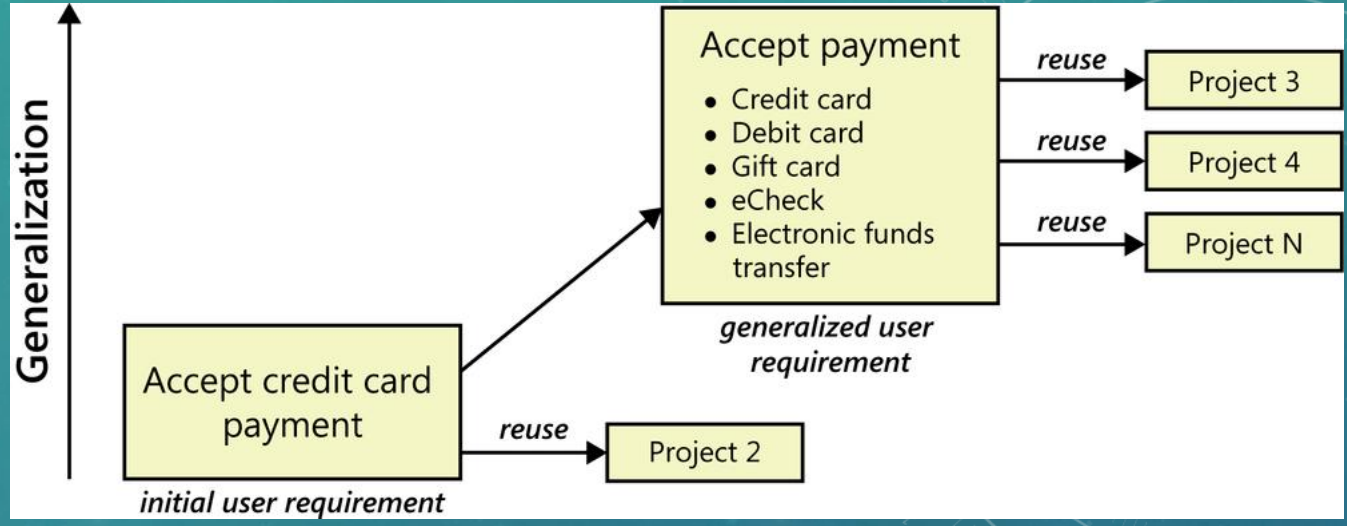
The diagram on the right side is a template showing the categories of information for each of the common types of requirements a project might encounter. Different types of requirement patterns will have their own sets of content categories.

A requirement pattern contains several sections:

- Guidance -- related patterns, situations to which it is (and is not) applicable, and a discussion of how to approach writing a requirement of this type.
- Content -- itemised explanation of the content that such a requirement ought to convey.
- Template -- requirement definition with placeholders wherever variable pieces of information need to go.
- Examples -- one or more illustrative requirements of this type.
- Extra requirements -- additional requirements that can define certain aspects of the topic, or an explanation of how to write a set of detailed requirements.
- Considerations for development and testing -- factors for developers to keep in mind when implementing a requirement of the type specified by the pattern, and factors for testers to keep in mind when testing such requirements.

	Software Requirements Pattern Template
Pattern Name	<descriptive name>
Pattern Id	<unique identification number>
Pattern Description	The system shall store the information corresponding to <relevant concept>
Author	<author name>
Source	< Source of requirement >
Classification	{Functional/ NonFunctional Requirement}
Classification-Purpose facets	<Type of Functional or Non functional requirement>
Goal	<Gives the purpose of the requirement pattern>
Applicability	< Description of area in which the pattern may be applied.>
Constraints	<Relevant Constraints if any>
Content data-Specific	<Specific data about the relevant concept> ….

MAKING REQUIREMENTS REUSABLE



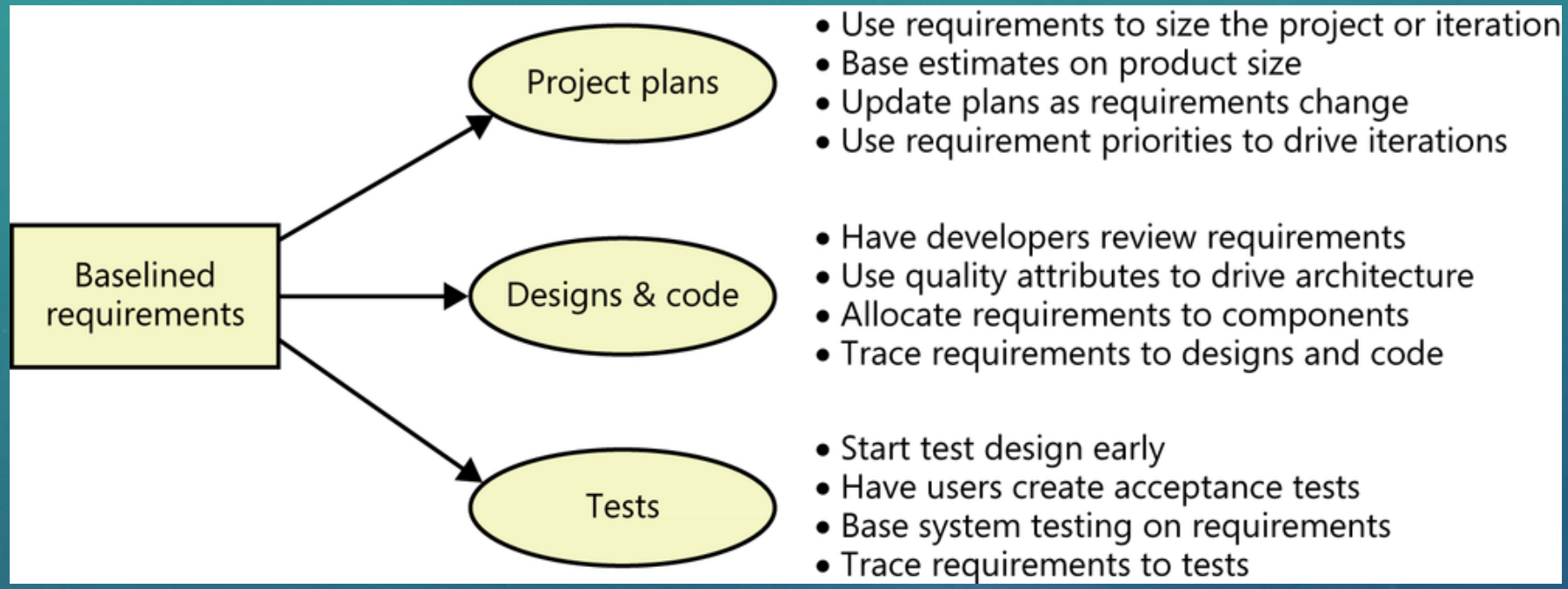
Key is the right level of abstraction

- Reusable requirements -- higher level of abstraction
- Domain-specific requirements – low level
- Example (see the diagram):
  - An application that includes a user requirement to accept credit card payments. This user requirement would expand into a set of related functional and non-functional requirements around handling credit card payments
  - Other applications might also need to take payments by credit card, so that's a potentially reusable set of requirements
  - Generalise that user requirement to encompass several payment mechanisms: credit card, debit card, gift card, and electronic money transfer
  - The resulting requirement offers greater reuse potential in a wider range of future projects. One project might need just credit card processing, whereas others require several of the payment processing methods
  - Generalising an initial user requirement like this—from "accept credit card payment" to "accept payment"—could be valuable even on the current project.

PART THREE: PROCEED FURTHER FROM REQUIREMENTS (CH19)

WHERE TO PROCEED?

Translate software requirements into rational project plans, robust design and test (three directions)



ESTIMATING REQUIREMENTS EFFORT

- Aim of this is to find out the percentage of effort should be used in requirement with respect to the total work in a software development project. (HR planning is an important part of project planning. This estimation tells the number of BA and their hours required.)
- Some figures:
  - A survey on 15 telecom and bank projects shows that the most successful projects spent 28 percent of their resources on requirements. The average project devoted 15.7 percent of its effort to requirements engineering
  - NASA projects that invested more than 10 percent of their total resources on requirements development had substantially smaller cost and schedule overruns than projects that devoted less effort to requirements
  - In a European study, teams that developed products more quickly devoted more of their schedule and effort to requirements than did slower teams, as shown

	Effort devoted to requirements
Faster projects	14%
Slower projects	7%

ESTIMATING REQUIREMENTS EFFORT

- Method -- three estimates (proposed by Seilevel, a consulting company)
  - First – percentage of total work, 15% is a normal consideration
  - Second – developer-to-BA ratio, 6:1 is the default value though some projects use 3:1
  - Third – activity breakdown, the hours that a BA spends on a number of activities such as user stories, reports, etc.
  - Requirements effort = *f*(first, second, third). (What is the function f? The authors do not tell us. They have a spreadsheet tool you may be able to access.)

FROM REQUIREMENTS TO PROJECT PLANS

Estimating project efforts

- Size depends on :
  - The number of individually testable requirements
  - The number of functions
  - The number of user stories or use cases
  - The number, type, and complexity of user interface elements
  - The estimated lines of code needed to implement specific requirements

FROM REQUIREMENTS TO PROJECT PLANS

Project scheduling

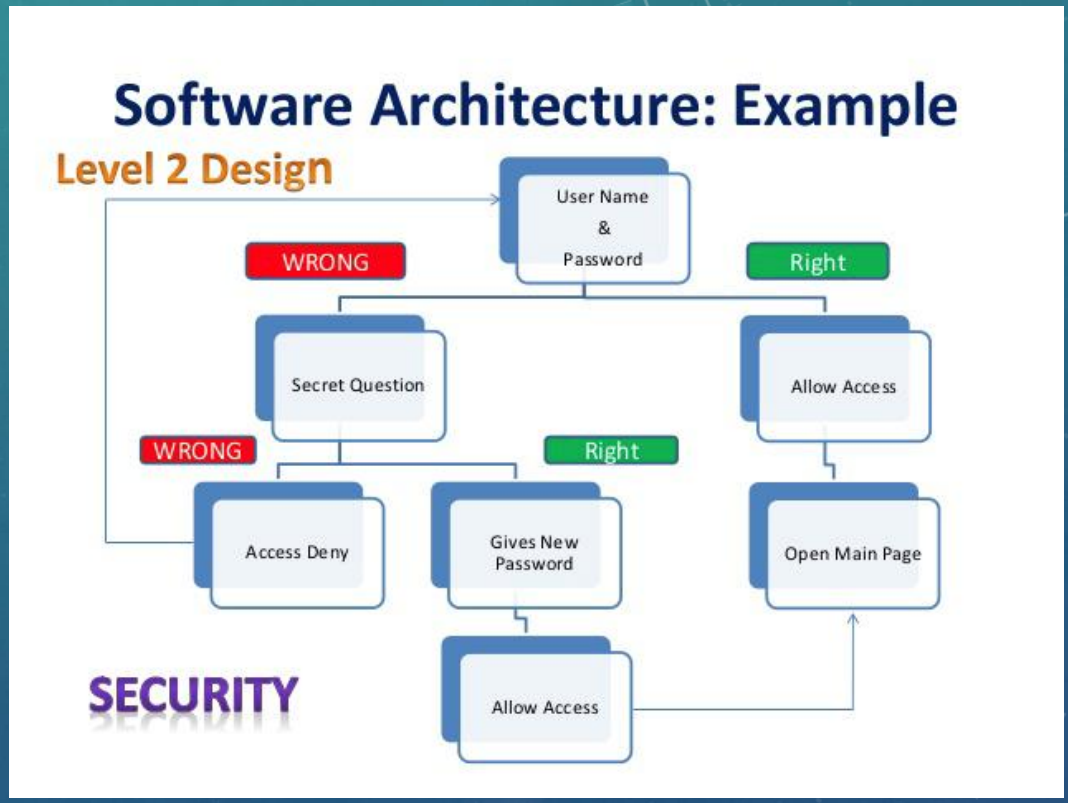
- Effective project scheduling requires the following elements:
  - Estimated product size (as seen on the last slide)
  - Team size (BA, developers, etc)
  - Known productivity of the development team, based on historical performance
  - A list of the tasks needed to completely implement to verify a feature or use case
  - Reasonably stable requirements, at least for the forthcoming development iteration
  - Experience, which helps the project manager adjust for intangible factors and the unique aspects of each project.



FROM REQUIREMENTS TO DESIGNS AND CODE

Architectural design

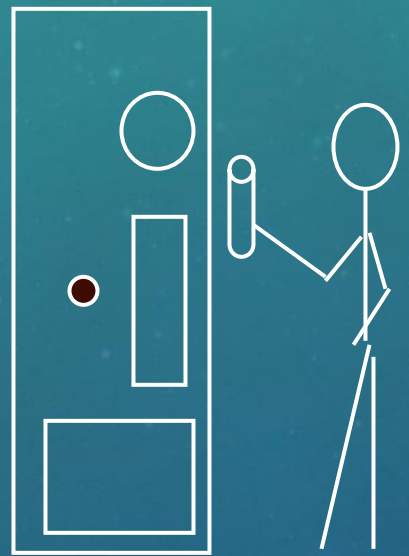
- It is the process for identifying the sub-systems making up a framework for subsystem control and communication. The output of this design process is a description of the software architecture. Architectural design is the early stage of the system design process.
- It allocates the high-level system requirements to the various subsystems and components
- It allows the development team track where each requirement is addressed in the design.
- The system requirements drive the architecture design, and the architecture influences the requirements allocation.

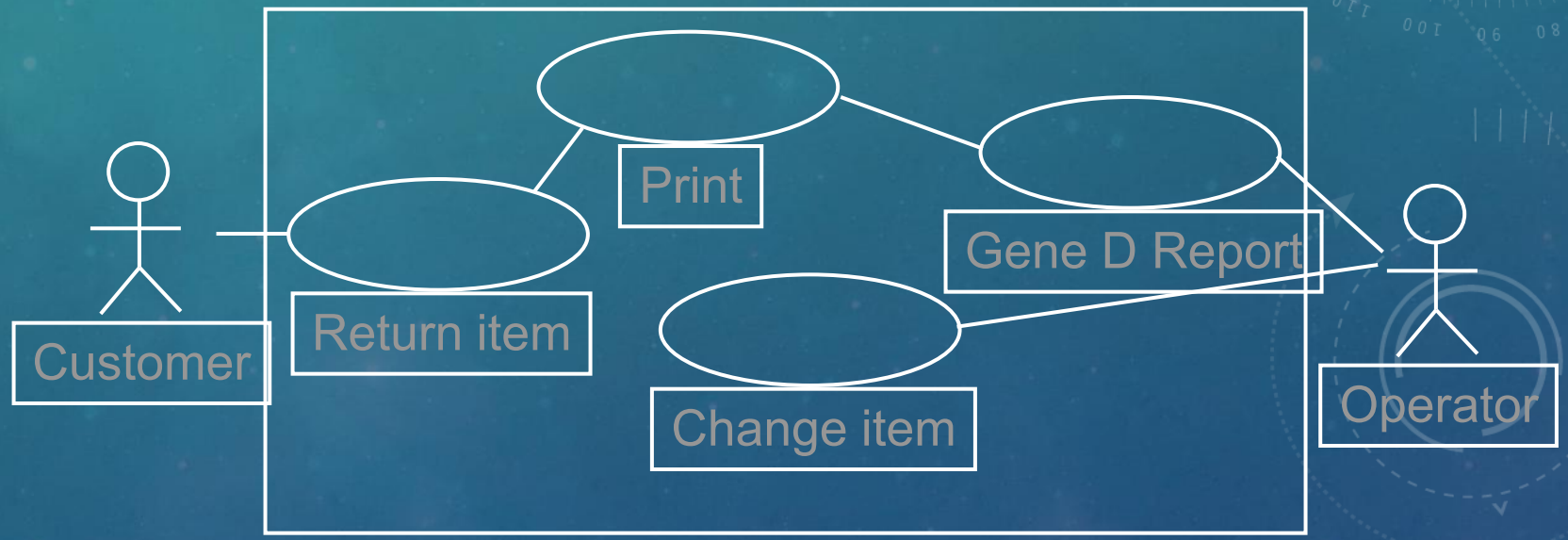


FROM REQUIREMENTS TO DESIGNS AND CODE

Software design

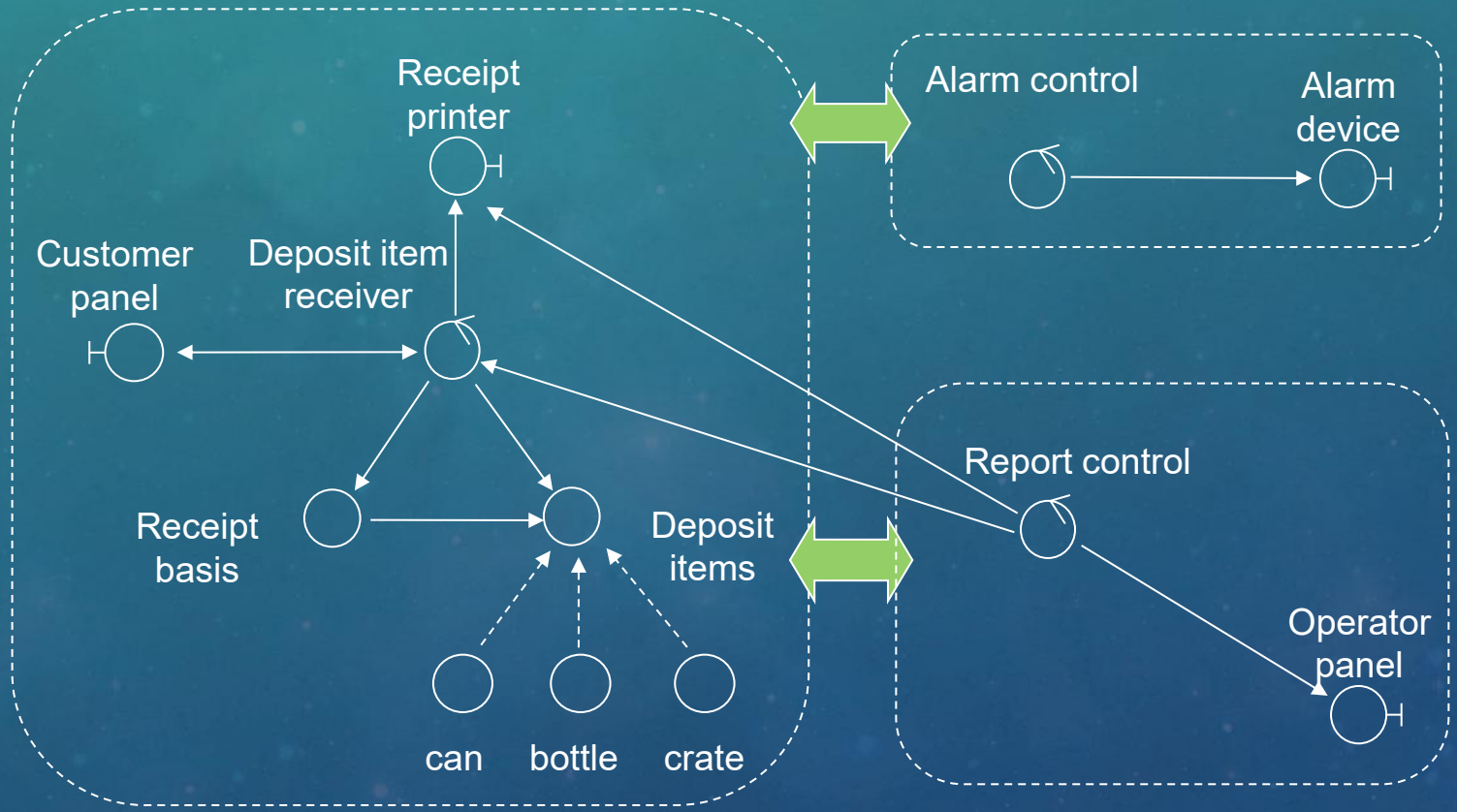
- Object-oriented analysis of requirements leads to class diagrams and other UML models. A designer can elaborate these conceptual class diagrams into more detailed object models for design and implementation.





FROM REQUIREMENTS TO DESIGNS AND CODE

The staff has to check, for each item, what type has been returned. He will register how many items each customer returns and when the customer asks for a receipt, he will print out what was deposited, the value of the returned items and the total return sum that will be paid to the customer. He also print out the total number of items that have been deposited at the end of each day. He has right to change the deposit values of the items through a console. When anything wrong with the machine, he will be called by a special alarm signal.



FROM REQUIREMENTS TO TEST

- The requirements are the ultimate reference for system test and user acceptance test, that is, “the product should be tested against what it was supposed to do but not its design and code” – this surely means validation rather than verification.
- What to test?
  - Expected behaviour under normal conditions
  - Expected behaviour under abnormal conditions
  - Quality attributes
- How to test?
  - Black-box
  - What about white-box test? – It is used to test design and code! However, it is useful to identify failures of black-box test.
  - White-box tests are based on logic in design, such as controls, and logic reasoning