ASE 285 Team Project

Software Engineering Rules and Tools

Dr. Samuel Cho, Ph.D.

NKU ASE/CS

- We have discussed (a) software engineering and (b) the toy project so far.
- In this section, we understand software engineering again by using concrete ideas, tools, and rules for the team project.
- Students use software engineering rules and tools to manage complexity to build software products on time and within budget in a team.

1. Software Engineering
2. Software Process
3. Requirements
4. Architecture/Design
5. Implementation
6. Software Testing
7. Deployment and Maintenance
8. Integral to All Phasese
9. Project Rules and Management

Software Engineering

- Do you remember the Ariane 5 rocket exploded shortly after the liftoff?
- Do you remember the recent Southwest scheduling catastrophe that stopped most Southwest flights?
- We have seen these issues over the years, and we will.

- Building software products is so hard that many software projects turned out to be failures multiple times in history.
- They are all from software issues, not hardware.

- It is easy to analyze the cause of the errors and say “How dumb they are!”
- However, if anyone is in their situation, they will likely make the same (or worse) mistakes.
- What is the main reason why software itself or building software products is so hard?
- How can a group of smart people repeatedly make such huge mistakes?

The problem: Complexity

- A mismatch in floating point numbers causes the Ariane 5 rocket disaster to cause a simple 'overflow exception' ($370M).
- The Southwest disaster is caused by neglecting to fix small bugs, leading to a total catastrophe ($220M loss).
- The software engineers could not fix the issues due to the complexity of the software system they were dealing with.

The Solution: Rules and Tools

- The Ariane 5 rocket disaster could have been prevented if they had used a simple tool, such as an integration test, or followed the rule that enabled them to find the bug.
- The Southwest schedule disaster could have been prevented if they had an effective tool, such as a software process, or followed the rule that forced them to improve the software system iteratively.

Software Engineering: Definition

This course defines software engineering as "rules and tools to build software products in a team."

- Software engineers and teams should set up and follow the rules.
- Software engineers and teams should use high-level and automatable tools.

- Software engineers and teams should be able to build the system; to do that, they should be able to design it.
- Software engineers and teams should be able to design and build products that clients will pay to use; to do that, they should be high-quality.
- Software engineers and teams should be able to design and build products in a team; to do that, they should use team rules and tools.

Application

Notice: *The items in the Application boxes are the actions that students should do in the team project.*

Application of SE - SE rules

1. Before the project, each team should make team rules.
2. In the team rule, each team member understands the penalty when they do not follow the rules.
3. Students keep adding and applying the SE rules to be professional software engineers.

Application of SE - SE tools

1. Before the project, each team decides what tools to use.
2. Each team member uses the tools for team-related activities.
3. Students should master using SE tools to be professional software engineers.

Software Process

Software Engineering as a Business

- We, software engineers, sell our software products or services to our clients.
- The clients can be the stakeholders, software users, or any entity who requests the software.
- We promise to deliver the high-quality products that meet clients' requirements.
- We design and build the promised products using rules and tools to deliver them on time and within budget in a team.
- This is a serious but lucrative business.

Possible SE Scenario in a Real-World

You work for a software company that builds and delivers software products as a contractor from clients, and you are a senior software engineer or an architect.

1. Clients request us about the software system they need.
2. You translate the problem domain into the solution domain.
3. You design the overall structure of the software system; you identify the information flow.

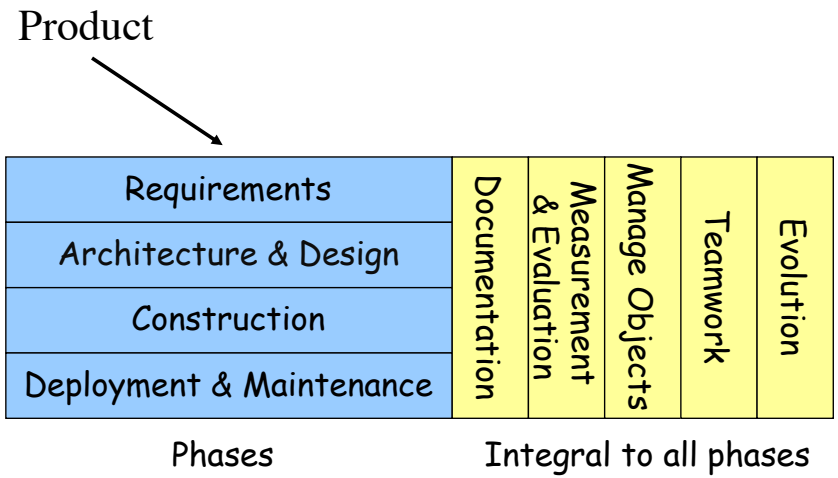
4. You divide the system structure into smaller modules iteratively until you can define a module with one responsibility.
5. You assign junior software engineers to implement(coding and unit-tests) the module.
6. You assemble the modules and check the information is flowing as designed.
7. You make sure the junior software engineers make and pass the tests to be sure the modules function as expected.

8. You make acceptance tests to prove that the required features are implemented.
9. You make regression tests to check if any changes won't break the existing code base; or if any changes that break the existing code base are identified immediately.
10. You deliver the software product to clients.
11. You make documentation for clients to use.

Warning: *This is a too simplistic view of software engineering, so we will add more detailed actions in building ASE/CS team projects and real-world apps.*

Software Process

- Software engineers invented software processes to manage these activities into (a) phases and (b) integral to all phases.
- The goal is to build and deliver high-quality products to clients.



Application of Software Process - SP

1. Students are required to understand (a) the problem domain, (b) the technology to solve the problem, and (c) the rules and tools for a team before the project.
2. Students should make requirements by interpreting the problem domain.
3. Students should make tests that match each requirement to prove that the features are implemented and working correctly.
4. Students should make documents for clients (we call them manuals) and other software engineers (we call them design documents).

The Agile Process

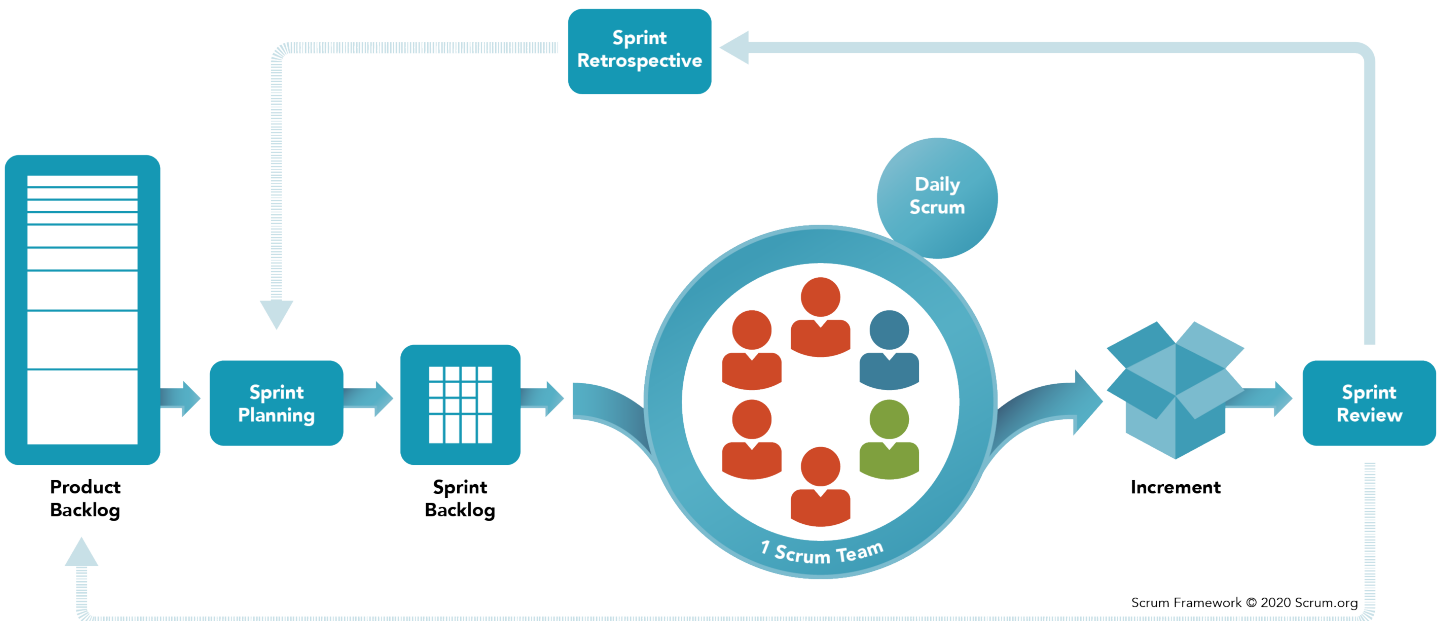
- Software engineers realized that having a big design and implementation is difficult, sometimes impossible, as "we don't know what we don't know" most of the time.
- So, they invented the Agile process to execute the software process in short intervals multiple times.

Application of the Agile Process - Agile

1. We use the Agile process in ASE courses.
2. We have three sprints this semester (sprints 0,1, and 2).
3. The first sprint (sprint 0), making things ready or prototype, is finished by building two apps, todoapp 1 and 2, that implement the CRUD operation.
4. In sprint 1, students (a) understand the architecture and design of the two todoapps and (b) make tests.
5. In sprint 2, students build their APIs with tests.

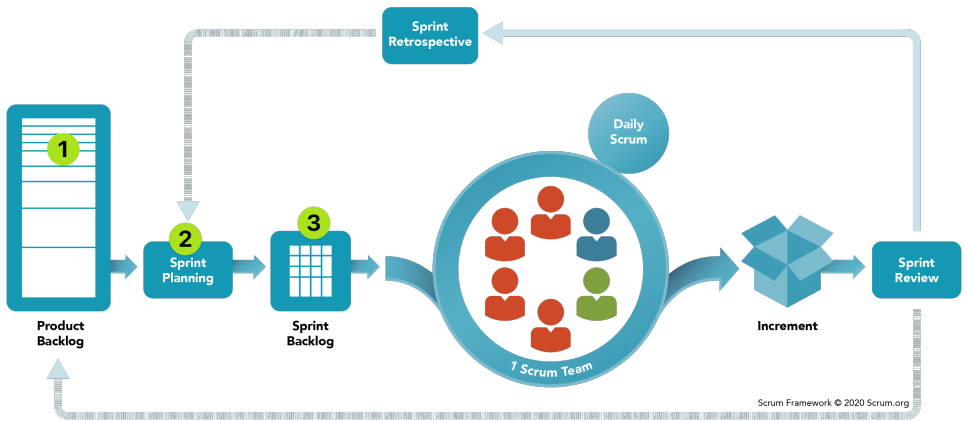
The Scrum Framework

- Specifically, we use the Scrum framework.
- Students must familiarize themselves with the Scrum ideas and terminologies.



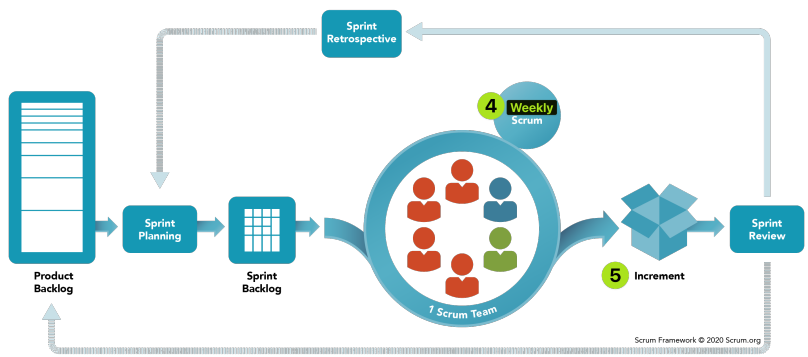
Application of Scrum - Scrum

1. Each team should have a product backlog before sprints.
2. Before each sprint, each team chooses what should be done in the next sprint.
3. Each team has a sprint backlog that will be burnt down during the sprint.



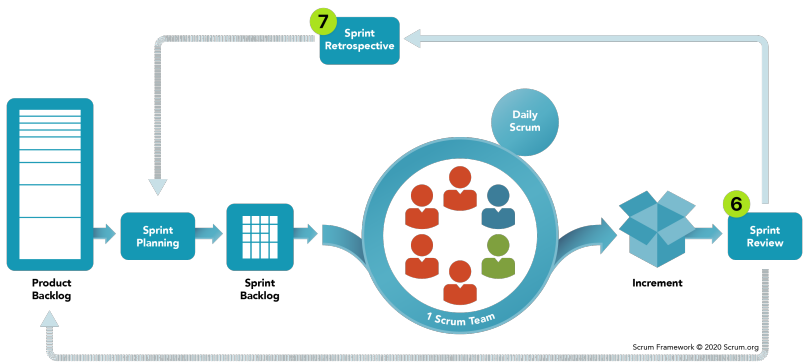
Application of Scrum - Scrum

4. We have a weekly scrum meeting to discuss (a) who burned down what tasks and (b) how many tests are made to verify (a).
5. We keep making increments (code, tests, documents, and such) and uploading them in the VCS (version control system) tool.



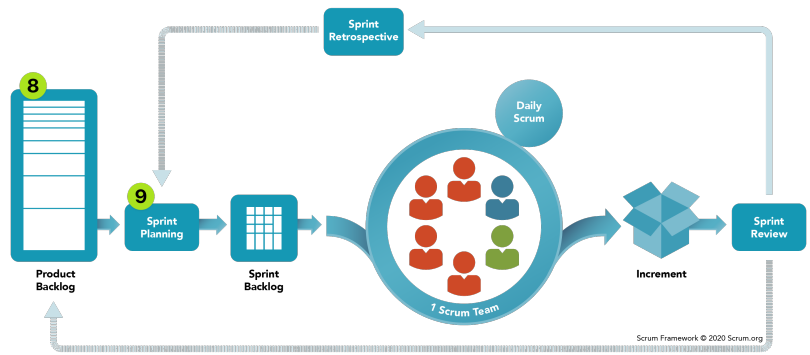
Application of Scrum - Scrum

6. At the end of the sprint, teams have a sprint review meeting to show the progress (often with demonstrations); they should show the burndown rates and test counts.
7. THen, each team should have a sprint retrospective to discuss (a) what went wrong, (b) what went right, and (c) how to improve.



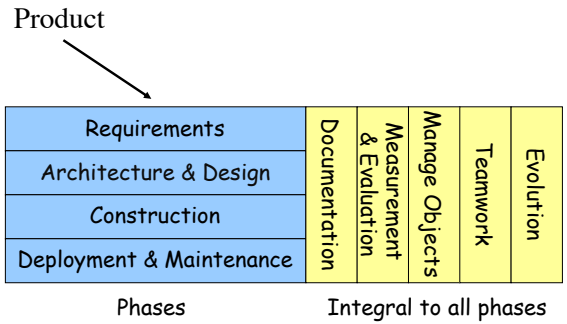
Application of Scrum - Scrum

8. After the sprint retrospective meeting, each team updates the product backlog.
9. In the sprint planning meeting, each team selects what should be burnt down in the next semester by whom and until when.



In the Next Chapters

- We discuss all the rules and tools for each activity (Requirements, Design, Construction, and Deployment/Maintenance) in the SE Box.
- Testing is a part of 'Construction,' but it is so important that we make a chapter for Tests.
- We also discuss the 'integral to all phases' rules and tools.
- Then, we discuss the project



Requirements

1. Clients request us about the software system they need.
2. You translate the problem domain into the solution domain.

Clients

- Clients request software building and delivery.
- Clients can be any entities, including (a) any people/companies that request software products for them, (b) managers who plan to develop and ship software products in the company, or (c) yourself when you have an idea to start a software company or open source projects.
- We assume cases (a) and (b), meaning clients generally do not know software engineering.

What Clients Want?

- Rumsfeld's law is used in this case: they don't know what they don't know (in most cases).
- Clients give software engineers only vague ideas as they don't know exactly what they want (in most cases).
- They can give us useful/effective feedback mostly from some concrete examples, so we need to show them examples to have some meaningful discussions.

Prototype

- So, when we don't know what we don't know clearly, the best way to solve this issue is to make an example that we call a prototype.
- A prototype is a working example to be on the same page with clients.
- A prototype is also a good way to understand (as much as possible) what we did not know before we dive into the project.

- When we build a prototype, we have a concrete example to show clients the possible product, which increases the possibility of success.
- We meet with clients to make documents on what clients need to be delivered; we call them 'requirements.'
- ***Warning:*** *We may not have the opportunity to make a prototype; in this case, we make requirements simply from the interpretation of clients' requests.*

Application Requirements - Prototype

1. Ideally, we should make a prototype before we start a project, but the professor's examples in sprint 0 (todoapp 1 and 2) replace the efforts of making a prototype.
2. Each team knows the project's goal of delivering new todo app APIs, so they can make a list of APIs they build and deliver before sprints 1 and 2.

What are requirements?

Requirements are tasks (features) to build and deliver to meet clients' demands.

- When we need some high-level requirements, we call them epics.
- We use user stories to express requirements.
- Each requirement should have a matching tests we call acceptance tests.

User Stories as Requirements

- There are many ways to describe clients' requirements.
- In the agile process, we use 'user stories' to describe or express requirements.
- A user story contains three pieces of information, A(who)/B(what)/C(why), as in line 1.

Code #User Story with Three pieces of Information

```
1 As a [A], I want to [B] so that [C].
```

Epic

- The user story specifies the clients' requirements by specifying (a) who, (b) what, and (c) why the requirement are needed.
- The course requirement can be described using the user story format as follows.
- This level of requirement is somewhat high-level, so we call it an epic.

Code #

```
As an [ASE 285 student],
I want to [learn software engineering rules and tools]
so that [I can use them in developing applications.]
```

In this epic example (E1), the professor can understand how to make the ASE 285 course from the information to meet students' requirements.

Who : Students.

What : They want to learn SE rules and tools.

Why : They can use them to develop applications.

Code #

```
E1: As an ASE 285 student, I want to learn software
    ↪ engineering rules and tools to use them in developing
    ↪  applications.
```

Two Requirements from Epic 1

- This high-level epic can be decomposed into two low-level requirements.

Note: *In the real world, we need to define the definition of 'developing applications' and decompose them into multiple actions, but we skip it to simplify*

Code #

```
1 E1R1: As an ASE 285 student, I want to learn software
    ↪ engineering rules so that I can use the rules for
    ↪ developing applications.
2 E1R2: As an ASE 285 student, I want to learn software
    ↪ engineering tools so that I can use the tools for
    ↪ developing applications.
```

Tasks as Sub-requirements

- Depending on tools or rules, the requirements can be divided even further to make tasks.
- It is up to each team to use tasks as sub-requirements or use only epics and requirements.
- For the rest of this document, I use the terms tasks/requirement interchangeably.

Tests - Acceptance

How can we know the professor built and delivered the course that meets the epic and two requirements?

- There is only one way to prove it - tests.
- For an epic, a matching test should be provided and passed to make clients can accept the product - we call it an acceptance test.

Code #

```
1 Acceptance Test for E1:
2 This test checks that students learned SE rules and tools
    ↪ and they used them in building applications.
```

Tests - Unit/Integration/Others

- For requirements, the same idea applies; depending on the nature of the requirements, we should make unit tests (lines 1 – 2), integration tests (lines 4 – 5), or any other tests to prove that the requirement is

Code #

```
1 Unit Test for E1R1:
2 This test checks that students understand SE rules.
3
4 Integration Test for E1R1:
5 This test checks if the rules are used to build an
    ↪ application.
```

Architecture/Design

3. You design the overall structure of the software system; you identify the information flow.
4. You divide the system structure into smaller modules iteratively until you can define a module with one responsibility.

Software Architecture

- We build a prototype to show the possible output to clients.
- We have requirements to express what should be implemented and tested for clients to accept the product.
- When we have requirements, we can design how to (a) implement the requirements through construction and (b) check if the implementation meets the requirements through testing.
- High-level design is called 'software architecture,' and low-level design is called 'software design.'

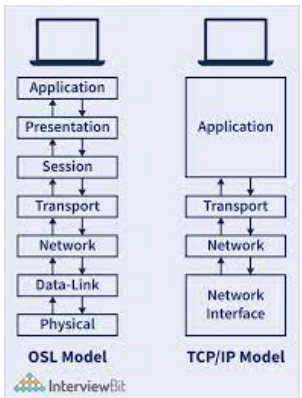
Definition of Software Architecture

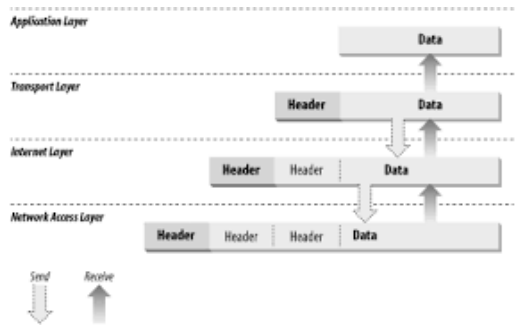
- We use the definition from Dr. Dewayne E. Perry.
- Software Architecture is concerned with selecting (a) architectural elements, their (b) interactions/data flow, and the (c) constraints on those elements and their interactions.
- We need them to provide a framework to satisfy the requirements and serve as a basis for the design.

SA = EFR

- In short, Software architecture (SA) is about (1) Elements, (2) Form (Interactions/Data Flow among the Elements), and (3) Rationale (Constraints on Elements and Interactions).
- In other words, Software Architecture (SA) = {Elements, Form, Rationale (EFR)}.
- Students can use a mnemonic "SAFER" instead of SA=EFR.

- The diagram on the left side shows the OSI Model and TCP/IP architecture.
- It uses rectangles to express each element, arrows to show the form (and the data flow), and the configuration of rectangles and arrows to show the rationale (and constraints) behind the structure of elements.
- In most cases, we may need to show the data structures each element uses to process the data flow, as in the right-side diagram.





Software Architecture for Visualization

In practice, we need software architecture (SA) to visualize the following:

Element: An element in SA should visualize the components to be designed; the element can be decomposed into sub-components until they have only one responsibility.

Form: The form should visualize the dataflow using arrows; the dataflow reveals the interaction among elements.

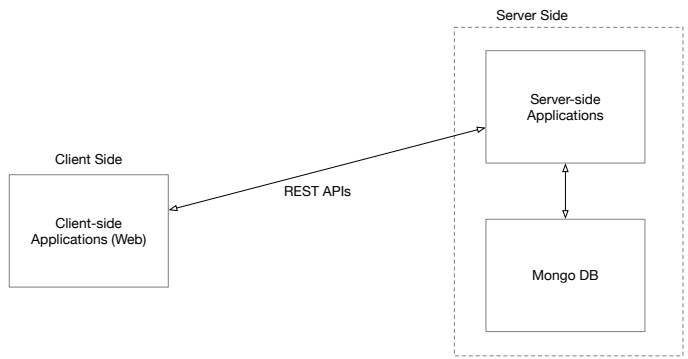
Rationale: The rationale using the structure of the form should visualize the reasons why some elements are forming the configurations.

This SA diagram shows the possible software architecture of the todoapp.

Element: There are three components (client-side app, server-side app, and MongoDB).

Form: Client-side app interacts (data flows) only with the server-side app using REST APIs.

Rationale: Server side has two components (constraints with dotted lines), and they are connected to interact and dataflow; MongoDB is separated from the client-side app.

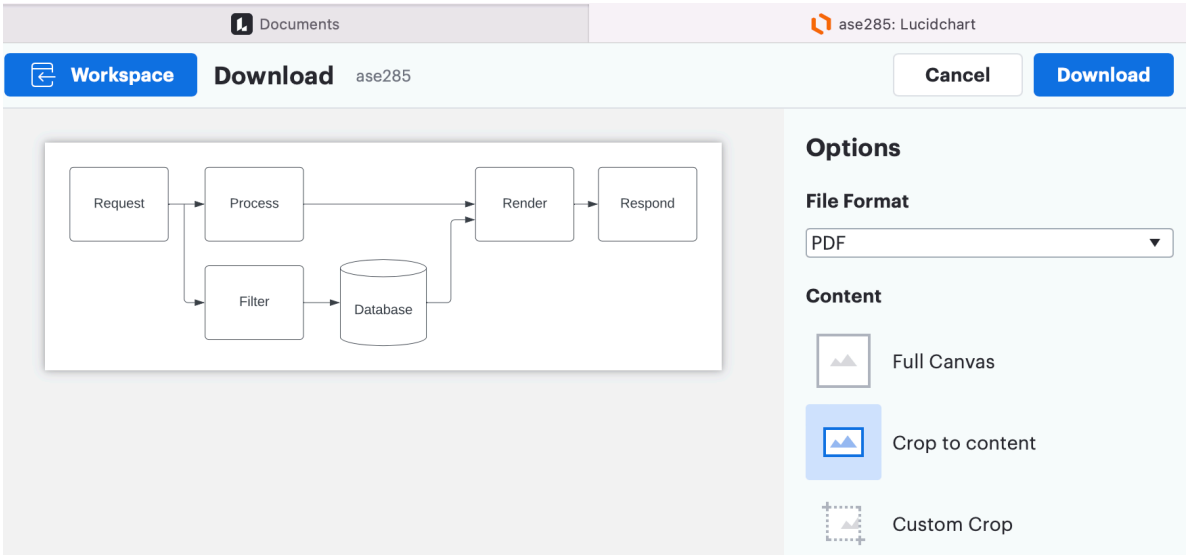


Application Software Architecture - SA

1. Each team should finish the software architecture diagram before sprint 1.
2. The software architecture diagram is an important part of software documents and deliverables.
3. The previous SA diagram is too simplistic, so each team should make diagrams with more elements, forms, and rationales.
4. The SA diagram should clearly show the elements, data flow/interaction, and constraints.

Diagram Tools

- For commercial tools, Visio (for PC) and OmniGraffle (for Mac) are the most widely used diagram tools.
- For free online tools, we can use Lucid Web App.
- The following diagram shows the Lucid app and the toy project architecture made with Lucid.



Software Design

- Software design is about (a) modules and (b) interfaces.
- Elements in software architecture should be decomposed into modules.

SRP and Unit-tests

- Each module should have only one responsibility; it is one of the software design principles called 'Single Responsibility Principle (SRP).'
- Each module should be checked if the input produces the expected output, and we call it unit-tests.

Application of Unit Testing

1. When students make modules (classes or equivalent), they should make unit tests to check the module's functions are correctly implemented.

Modules and Integration tests

- Each module should interact with other modules, and data/information flows through them.
- These actions should be tested, and we call these tests 'integration tests.'

Application of API Testing

2. Testing APIs is practically doing the integration test between a client and a server.
3. Students must make API tests for the integration tests.

ASE 420

- Software Design is one of the core Software Engineering activities.
- We have a Software Design course (ASE 420) to discuss software design topics exclusively.
- In the course, we discuss (a) OOP, (b) design principles, (c) refactoring, and (d) design patterns.

Implementation

5. You assign junior software engineers to implement (coding and unit-tests) the module.
6. You assemble the modules and check the information is flowing as designed.

Implementation and Tests

- Implementation is an activity to translate the design into a specific programming language.
- The software architecture is laid out: (a) elements, (b) form, and (c) rationales are defined.
- Each architectural element is decomposed into modules and the information flow and interaction among modules are defined.
- Each module is well-designed with a single responsibility.
- So, implementation (coding and testing) becomes a somewhat mechanical process at this stage.

Junior Software Engineer's Role

- As a result, most junior software engineers contribute to their company by implementing code and making tests.
- They should master coding and testing skills, but they should also know how to design software to be ready for high-level roles such as senior software engineers.

Application of Coding/Testing

1. Each team leader is responsible for managing coding and testing; they make tests with team members.
2. Each team member is responsible for implementing the features (and their unit-tests) they promised to deliver.

Software Testing

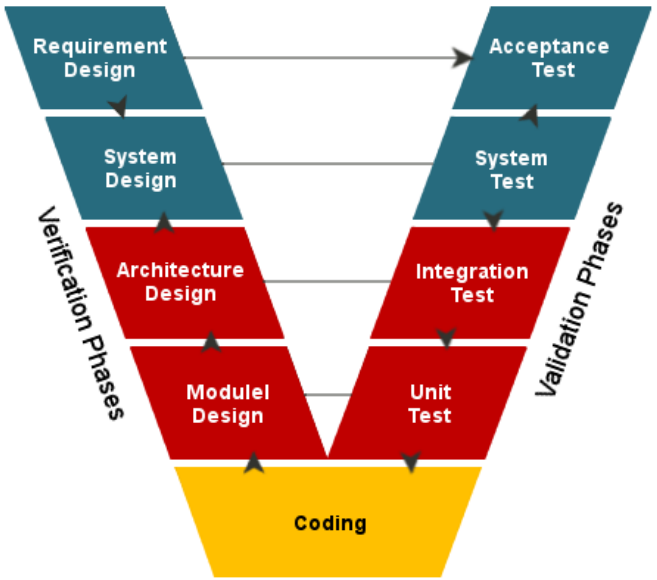
7. You make sure the junior software engineers make and pass the tests to be sure the modules function as expected.
8. You make acceptance tests to prove that the required features are implemented.
9. You make regression tests to check if any changes won't break the existing code base; or if any changes that break the existing code base are identified immediately.

What to Test?

- We test if the implementation meets the requirement: acceptance tests.
- We test if the software system behaves correctly or produces expected output: system tests.
- We test if data flows among modules as expected: integration tests.
- We test if functions in a module produce expected output with given inputs: unit tests.
- We test if the added/updated code does not break the existing system: regression tests.

Software Process for Tests

- The V-model is a well-known software process model focusing on software testing (CSC439 topic).
- In this model, we both validate (check if we do the right tests) and verify (check if we do tests right).



Application of Tests

1. Team leaders should make acceptance tests and integration (API) tests.
2. We usually do not make system tests unless we have special system features to test.
3. Anyone who makes the component should make unit-tests, and team leaders manage/help them.

JavaScript Jest Unit/Integration test Framework

- JavaScript provides a unit test framework Jest.
- Let's say that we have a module (sum.js) that has a sum function.

Code #

```
1 function sum(a, b) {
2   return a + b;
3 }
4 module.exports = sum;
```

We can test this unit (module) using Jest.

1. Install Jest with ‘npm install jest.‘
2. Make a unit test file (sum.test.js) that checks the functions in the module sum.js.
3. Notice that the format of the Jest is `test("Description", f)`, and the function f should compare the output (expect) and correct value (toBe).

Code #

```
1 const sum = require('./sum');
2
3 test('adds 1 + 2 to equal 3', () => {
4   expect(sum(1, 2)).toBe(3);
5 });
```

- Then, add the following description to the ‘package.json’.
- Run ‘npm test run‘ to get the unit test results.

Code #

```
1   "scripts": {
2     "test": "jest"
3   },
```

```
smcho@mbp unittest-examples> npm run test

> test
> jest

 PASS  ./sum.test.js
  ✓ adds 1 + 2 to equal 3 (3 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.424 s
Ran all test suites.
```

GUI Testing

Application of GUI testing

1. The web application development should include GUI testing, which is hard to automize.
2. Each team should do the GUI testing manually, and at the same time, they should do the GUI testing indirectly.
3. For example, to test the list API, we may need to use a web browser to get the HTML and check if the information is correctly transferred.
4. Or we can make and test listJson API to get JSON API instead and make the list API to use the listJson API.

Regression Testing

Application of Regression testing

1. Testing should be as automatic as possible, and regression tests execute all the testing code (unit, integration, and GUI).
2. When some code is added, or the existing code is updated, regression tests should be executed to verify that nothing is broken from the change.
3. Making regression tests is software development, and team leaders are responsible for making regression tests.

Note: *There are many tools for automizing GUI tests, but using them is not trivial in most cases. So, at ASE 285, students don't need to automize GUI testing.*

Application of Regression Testing

4. Students should do GUI testing (a) manually and (b) with an indirect method, as explained.
5. or manual GUI testing, each team should have documents for the process, and for regression tests, they should use indirect GUI testing instead.

Todo App Acceptance Test Example

- From the todoapp 1 example, we implemented the (simplistic) Create operation with the 'add' APIs.
- This is from the client's epic E1 (lines 1 – 2).
- We also need to define an acceptance test for E1 (lines 3 – 5).
- We have to give the link to the test code (line 6).

Code #Epic for the add feature

```
1 E1: As a user, I want to add (a) a todo item and
2   (b) the date so that I can record my todo item and date.
3 Test of E1: When a user uses the '/add' GET function from a web
4   browser with a form with (a) a todo item ("hello") and
5   (b) the date ("today"), the information is added to a server
6   and should display "OK."
7 Link to the test E1: ...
```

- Students can modify how the regression tests are implemented.
- For example, they can use CURL to give the same input to the server without using web browsers to automize the test.
- In short, we should prove that the epic that clients requested is implemented and tested.

- The epic E1 can be decomposed into smaller requirements.
- It is similar to the epic but written from a programmer's perspective; users do not need to know about technology such as MongoDB.
- It is also a part of the requirement to constitute an epic E1; for example, we need to specify how the information is shown to the users when the document is successfully added to the database.

Code #

```
1 E1R1: As a programmer, I want to make a todo API so that
    ↪ when I give (a) the todo item and (b) the date, the
    ↪ information is uploaded to MongoDB DB.
2 Test of E1R1: When a programmer uses 'add' API using a web
    ↪ browser with (a) a todo item ("hello") and (b) the
    ↪ date ("today"),  the MongoDB should create one
    ↪ document with (a) and (b).
3 Link to the test E1: ...
```

Todo App Integration Test Example

- We need to make an integration test for E1R1.
- In this case, we should make an API test for the integration test.
- There are many tools for integration tests; however, using unit tests is the most convenient.
- For API tests, there are many tools such as https://www.postman.com or the equivalent.

Todo App Regression Test Example

- When we finish the coding of the todo app, we have unit tests (from team members), integration tests (from team members and leaders), and acceptance tests (from team leaders).
- Team leaders should make regression tests — automize the execution of all the tests so that each team member can run the tests automatically when they modify the existing code base.
- Team leaders should automize as many GUI testings as possible for the regression test.

Application of Software Tests

1. Building high-quality software products using a variety of tests is one of the top priorities of the team leader.
2. A team leader should make acceptance tests.
3. A team leader should make integration tests.
4. A team leader should make regression tests.
5. A team leader and the implementer of the API should make API tests.
6. Team members should make unit tests.

Deployment and Maintenance

10. You deliver the software product to clients.

Deployment

- When the web application is built and passes all the acceptance tests, we should be ready to deploy the products.
- We can use (a) ngrok, (b) Heroku, or (c) cloud services such as AWS to deploy the web applications.
- We can also use GitHub when the product is an open-source program.

Maintenace

- To maintain the software products, we should make a bug-reporting system that (a) accepts the bug report from users, (b) record the bug in the database, (c) and assign the bug fix to software engineers.
- Building a system to maintain bugs is another software engineering project, so instead of building the system, we can use existing open-source or commercial tools.

👉 Application of Deployment and Maintenace

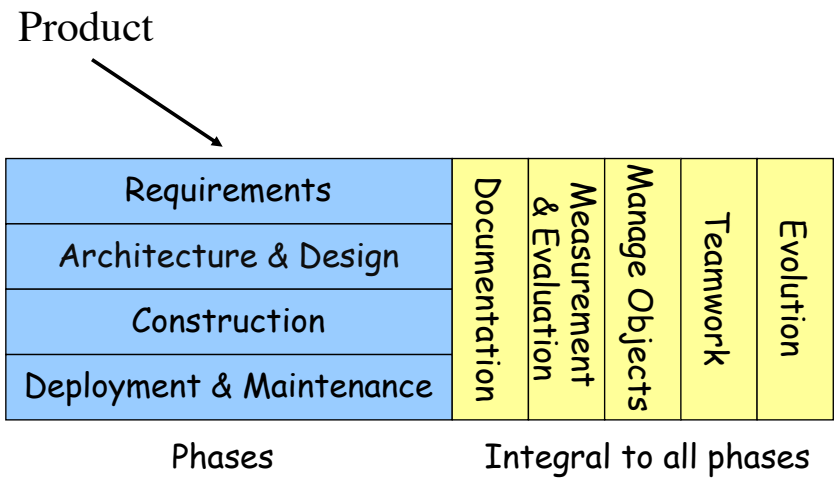
1. We do not make deployment or maintenance systems in this course.
2. However, we learn Git/GitHub so that students know how to use these tools for software maintenance.

Integral to All Phasese

11. You make documentation for clients to use.

Integral to All Phases

- We have discussed the software process phases starting with requirements and ending with maintenance.
- There are also other software process activities that are common to all phases.



Evolution and Manage Objects (MO)

Application of Managing Evolution

1. We use Git and GitHub to manage the evolution of projects and objects.
2. We will discuss these topics in the class meetings.

Teamwork

- All the engineering activities, including software engineering, are team-based.
- We, software engineers, are evaluated by peers and superiors all the time, so (a) enhancing problem-solving capability and (b) maintaining integrity are among the top priorities as software engineers.
- The detailed rules and tools for teamwork from a project perspective areag discussed in the later part of this document.

Measurement

- Students need to measure their progress; LoC (Lines of Code) is the easiest way to measure progress.
- Consider using https://github.com/AlDanial/cloc or any other tools for the measurement.

Warning: *We are problem solver, not coders. So the LoC is one of the tools to measure progress; Focus on solving problems with effective coding, not making a ton of code.*

Evaluation of A Team

- We, software engineers, evaluate each other all the time.
- Managers evaluate engineers, engineers evaluate managers.
- High-level people are evaluated by the CEO.
- The CEO is evaluated by a board and market.
- Fair evaluations guide us and encourage us to improve.

Application Measurement

1. Team members measure LoC (code and tests) to show their progress to team leaders before the end of each sprint; when a part of code/tests are shared among engineers, only one engineer reports the measurement results.
2. Team members add all the LoC to report at the sprint review meeting.

Application of Evaluation

3. At the end of the semester, team leaders evaluate team members.
4. Team members evaluate team leaders; they don't evaluate other team members, but they can report exceptional contributions or issues of others to the high-level managers (the professor).

Evaluation - An Agile Process

Application of Evaluation - an Agile Way

1. At the end of each sprint, we have a sprint retrospective meeting to evaluate a team's performance.
2. The team leader leads the meeting and makes a report about (a) what is done right (and why) and (b) what is done wrong (and how to improve the issue).

Documentation

There are two types of software engineering project documentation:

Manuals: are communication or promises between us (software engineers) and users.

Software Design Documents: are technical documents written by software engineers for other software engineers.

Manuals

Who Writes the Document?: Even though a company has technical writers, at least the draft should be written by software engineers. Writing manuals is so important that mostly high-position people, such as architects or managers, are responsible for the manual writing.

What Should be Written?: The manual should explain (a) the basic structure of the application, (b) the behaviors of each GUI component, and (c) any information to help users.

Design Documents

Who Writes the Document?: Any software engineers who architect or design a system or features should write the software design documents.

What Should be Written?: The design documents should explain (a) the architecture of the application, (b) the responsibility, inputs, and output of each module, and (c) any information to help software engineers.

Application of Making Documents

1. Team members make the manuals for their parts, but team leaders should be responsible for assembling each part to make high-quality manuals.
2. Team leaders should make the software design documents; however, in this course, they should make (a) architect diagrams and (b) (optionally) design documents only when the team designed the software.

Project Rules and Management

Team Rules

- Each team should start with the team rules.
- The rules should contain (a) general rules (b) communication rules, (c) progress report rules, (d) submission rules, (e) contribution rules, and (f) presentation rules.
- When we make rules, we should also specify the penalty; otherwise, no one would obey the rules.
- When anyone does not obey these general rules, they will be given 0 points (or deduct points) in the corresponding evaluation rubrics.

General Rules

These are guidelines, so each team can add more rules.

- Each team player (members and a leader) carries out the project activities as a professional, not a student.
- Each team player takes responsibility for their work.
- Each team player does their best to make high-quality products.

Communication Rules

- Each team member should understand that being a good communicator is essential as a professional software engineer.
- Each team member should respond to requests within 24 hours (excluding weekends).
- No surprise rule is applied to communication; whenever unexpected things happen, each team player should let others know as soon as possible.

Progress Report Rules

- Each team player must report their project progress using tools Canvas (or any other tools) for sprint review meetings and weekly scrum meetings.
- Each team leader must aggregate all the progress results and upload them on Canvas for sprint review meetings and weekly scrum meetings.

- This is an example of the report.
- When a stakeholder clicks the link, the link should lead to documents or any information that matches the tasks/requirements/tests.

Code #

```
Total tasks this semester
=========================
6 Epics (E1 - E6, link) and 6 Tests (link)
17 Requirements (E1R1 - E6R5, link) and 20 Tests (link)
5 Requirements assigned to Sam (link) and 7 Tests (link)
... (other requirements or tasks assigned to other members)

Total tasks this sprint progress
================================
Out of 3 Epics and 6 Tests, 1 Epics (33.3% burndown rate, link) and 3
    ↪ Tests (50% burndown rate, link) are finished.
Out of 17 Requirements ...
Out of 5 Requirements assigned to Sam ...
... (other requirements or tasks assigned to other members)

Total tasks this week (Spring 1 Week 2)
=======================================
... (same format)
```

Progress Report Tools

- Each team can use any team management tools (Notion, Trello, Asana, or Jira/Confluence).

Application Progress Report Rules

1. Before sprint 1 starts, team leaders should make the sprint and product backlog using the tool (with links so stakeholders can access them).
2. Before sprint 2 starts, team leaders should share what features team members implement with links using the tool.
3. The professor will be ready for the Canvas page so that each team can upload the plans (deadline and milestones) and progress

Application Progress Report Tools

1. Each team uses a Canvas page to share/report the progress; team leaders are responsible for the reporting, and team members should provide the information one day before the weekly scrum meetings.
2. The progress should show the numbers (a) how many tasks (or requirements) should be done this semester and this sprint, (b) how many tasks (or requirements) are finished (by each member), and (c) how many tests are made and tested.

Application Progress Report Tools

3. Each team leader uses Canvas to report all the details of their project progress; the Canvas page should have a link so that stakeholders can check the progress easily.
4. When a team member cannot report to the team leader (or submit the report late without proper communication), the team leader makes the team member's progress 0 (no burndown) and make a note to write the behavior in the evaluation report.

Application Progress Report Tools

6. Each team member should report their burndown progress with test coverage to the team leaders by the end of the day before the weekly scrum meeting or any presentation.
7. Each team leader collects all the burndown progress and test coverage to upload the information on Canvas before the scrum meetings or any presentation.
8. Each team leader **never** urges team members to send the report; it's one of their core responsibilities, and if they fail to do so, it's all their problem, not the team leaders'.

Submission Rules

- Each team player should submit the highest quality products possible as a professional.
- To accomplish this goal, each team player should start early to finish early.
- Any deliverables should bear the creator's or updater's name to show who is responsible for the submission.

Contribution Rules

- Each team player should understand that they should contribute to the success of the team; otherwise, the team does not need the player.
- Each team player is responsible for certain tasks and should deliver value by repeatedly finishing the tasks.
- Each team player should join forces to solve problems; when anyone in the team needs help, the team player should help them attain the common goal.

Presentation Rules

- Make a presentation from a stakeholder's (audience's) perspective: tell only what they want to know, don't tell what you want to say.
- Talk is cheap and dangerous, but showing is hard but safe; try not to talk but to show.
- When you don't have anything to say, say nothing and finish the presentation.

Do's in the Presentation

- Make it short when you do a presentation, but add more pages for the detailed questions.
- Make the presentation as simple as possible but give links so stakeholders can access the detailed information anytime.
- Always show pages in the slides so stakeholders can ask questions from the page information.

Do not's in the Presentation

No Scratching Heads: Be an actor as if you know everything when you present something; when you failed to do that, everyone remembers you as a dumb person. Also, don't argue with somebody and waste everyone else's time. Say, "Can we talk about this later in person?"

No Pointing Fingers: Never say "it's his/her fault" in the presentation because it is the team's fault.

No Emotions: Calm down and relax, be a professional, and act like a professional.

Presentation Examples 1

For weekly meetings:

- "In this week, we have burnt down 10% of the sprint requirements, and all the finished requirements are tested by 5 unit tests, 3 integration tests, and 2 acceptance tests. Sam, who is responsible for the front end finished ... "
- "You can access all the related information from the Canvas page ..."
- "We had an issue with MongoDB database access, when we ... we could not ..., we are working on it, and we plan to solve this issue by the end of next week." or "could you help us to fix this issue?"

Presentation Examples 2

For sprint review meetings: You are in trouble if you can't burn down all the requirements.

- "In this sprint, we have burnt down 100% of the sprint requirements; we also tested all the 4 epics and 10 requirements using 11 unit tests ... "
- "Let us show you some demo", screen capture, or video recording is OK.
- Remember to make it short and to the point; stakeholders are only interested in the progress (burndown rate and tests coverage) and output.

Presentation Tools

- For simple presentations, do not use big applications such as PPT or similar.
- For sprint reviews or weekly meetings, do not use animations or unnecessary fancy features; make it as simple as possible.
- Try to use https://marp.app that uses Markdown format for making HTML/PDF presentations.

CS vs ASE - Coding in ASE

- ASE is pretty different from CS, and now ASE 285 students can see it clearly.
- They need to do a lot of activities other than coding to build and deliver products.
- **Coding is important in SE but not a big part of SE.**
- If you think about it, any money-related business has the same characteristics.
- For example, in Hollywood movie making, acting is important, but not a big part of it.
- Also, playing music is important in the music business, but not a big part.

As a Professional

You are not allowed to say the following when you are a professional, as they can ruin your career:

It's confusing: Instead say, "This is my interpretation X, Y, and Z, are they correct?"

I didn't know: Instead, say, "I will check it out. Is there anything I need to know in advance?" or simply "I'll make sure about it."

I did my best: Please don't say this; nobody cares, and you look like an incompetent software engineer. If you want to say something, say "I will be better next time."