Module 2

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

Module 2 Study Guide and Deliverables

Module Topics:	• Topic 1: Requirement Analysis and Management Using User Stories • Topic 2: From Requirement to Design – UML Class Diagrams and State-Transition Diagrams.
Readings:	• Online lecture notes • Braude, Part IV (or related chapters in other textbooks)
Discussions:	• Weekly Group Meeting
Assignments:	• Lab 2 due **Tuesday, September 19 at 6:00 AM ET**
Assessments:	• Quiz 1 due **Tuesday, September 19 at 6:00 AM ET**
Live Classrooms:	• **Tuesday, September 12 from 7:00-9:00 PM ET** • **Thursday, September 14 from 7:00-8:00 PM ET**

Learning Outcomes

By the end of this module, you will be able to do the following:

- Describe and compare functional and nonfunctional requirements.
- Explain how to create high-quality user stories.
- Create user stories, break down tasks and acceptance tests, and manage user stories.
- Describe and compare entity, boundary, and control objects, as well as identify these objects from the requirements represented by user stories.
- Describe different class relationships and draw class diagrams.
- Conduct detailed requirement analysis for the group project in iteration 1.

Introduction

Before we can implement anything, we need to figure out the requirements. The requirements basically define what the customer wants to have in their software, either explicitly or implicitly.

Topic 1: Requirement Analysis and Management Using User Stories

Requirements analysis is the process of gaining the necessary understanding of requirements. The challenge is that customers usually do not know exactly what they want at the beginning. The development team needs to help customers figure out what they really want.

Gathering Requirements

Actually, not only customers, but all stakeholders in a project may be involved in the requirement-analysis phase. The first task is to gather requirements. There are different techniques that can be used.

- **Bluesky**—Simply brainstorming possible ideas with customers and other stakeholders. Make sure to get everyone involved. This may be not very effective without much guidance.
- **Use a list of questions**—Asking customers for a list of predefined questions can help in formulating ideas about the requirements.
- **Role-play**—Pretend to be the end users and imagine how they will use the system.
- **Observe the real-world application-domain example**—For example, if we wanted to create a student-registration system, we could first observe how registration was done manually, with paper and pen. It is important to understand the work flow of the job itself.
- **Review available documents**—If there are already some requirement documents about the system or the application, it is very important to review these first.
- **Observe current system**—If working on an existing system, observing the behavior of the current version and performing some basic testing can be a good way to identify old and new requirements.
- **Research similar systems**—Looking at similar systems can help customers identify what they do and don’t want, and define the differences between these examples and their own systems.

Examples

ProjectPortal

In this module, we will use the ProjectPortal project as an example. Project-based learning (PBL) is a student-centered pedagogy that integrates knowing and doing. It is a style of active, inquiry-based learning. Most (if not all) of our courses have project components. This course also features a semester-long group project. However, the courses' respective projects are isolated from each other. There is no connection between courses at the program level. Faculty and students are not aware of projects in different classes, even though they may be related or similar. To facilitate better communication between students and faculty members in different courses, we would like to develop an online project portal where students and faculty members can publish completed and ongoing project information to share with others.

Now that the idea is formulated, we can use the above techniques to figure out what we want from this tool. First, we need to research whether any similar tools exist. After some research, we find that, while quite a few project-related platforms exist—such as GitHub for version control, Pivotal Tracker or Jira for project management, etc.—there is no such tool that exactly matches our needs.

Then we can use bluesky or role-playing to gather requirements. For example, we can imagine that students or faculty members will publish their project information, and other students can then search projects by course number, program, subject, platform, skill, etc. They may mark projects as favorites and comment on them. Faculty members may provide feedback. They may also connect through the platform about possible collaboration.

Blackboard System

Another example we will use in this module to discuss requirement analysis is the Blackboard learning system, which we are very familiar with. This existing system already has a lot of functionalities.

User Stories

There are different ways to represent requirements. Usually, high-level requirements are simply listed in plain language in the project-vision document, without any specified format or template.

User stories are commonly used in Agile software development to represent software's functional requirements and drive the whole development process. A user story is usually written on a 3" x 5" index card and therefore must be very short—only a couple of sentences—so that it can fit. A user story should just describe one particular feature or requirement. It is easy to shuffle, add, or delete another index card. This makes user stories very flexible and easy to manage. User stories should be written in everyday language, without any technical jargon, so that they can be easily understood by customers. In fact, user stories are customer driven, written with or by and for customers, so that they can correctly represent what customers want. Other stakeholders—such as people from the marketing department, lawyers, HCI expector—may also participate, editing or adding

requirements based on marketing, regulations, and user experiences. As adapting to changes is one of the core values in Agile development, user stories are subject to change and can be easily reprioritized.

User Story Title and Description

Let us first look at some examples to see if they are good user stories:

Test Yourself

User interface (UI) needs to be very easy to use.

Yes

No

It is not good, because it is too vague to say “easy to use” or something like “user-friendly”. It means differently for different users. It needs to be specific. Furthermore, this is not a functional requirement, but a nonfunctional requirement.

Test Yourself

Use the Java Swing library for UI.

Yes

No

A user story should not use any technical jargon that the customers don’t understand. Java swing library is an implementation detail. It should not be listed as a requirement, unless the customers directly require it.

Test Yourself

Students can search for the project by the course number.

Yes

No

This is good. It clearly specifies who can do what.

Test Yourself

The faculty member can provide feedback on the project.

Yes

No

This is also good.

Here are a few examples of informal user stories for the Blackboard system:

- Students can submit their homework.
- Instructors can post lecture notes.
- Instructors can post grades for the students' assignment submissions.
- Students can check their grades.
- Instructors can post announcements, which can also be sent to students by email.

A user story should have a **title** so that it can be easily referenced. A title is usually an active verb expression that clearly summarizes the intended feature, such as "List Projects" or "User Login."

A **short description** follows to describe the type of user, what they want, and why. A commonly used template follows:

As…(an **end user**), I can/want to…(use some feature/do something), so that…(value, reason).

Examples

Add a Project

As a project owner, I want to publish my project to the platform so that other users can view it.

Another Example for the ProjectPortal Project

Search Projects by Course Number

As a user, I want to view projects related to a particular course so that I know what can be produced in this course.

Blackboard System Examples

Submit an Assignment

As a student, I want to be able to submit my assignment on Blackboard so that my instructor can grade it.

Post an Announcement

As an instructor, I want to be able to post an announcement to my whole class on Blackboard so that students are notified of what is going on in the class.

Test Yourself

The following examples are not good descriptions. Why?

1. As a user, I want to be able to edit the projects so that I can update the project information.
2. As a product owner, I want the system to be able to delete projects so that users have the option of deleting projects.
3. As a developer, I need to design the database schema so that I can implement it.

End-User Roles

To define the requirements, we should first have a good understanding of the potential users so that we can imagine what they want and how they will interact with the software system. Usually, a software system can be used by different types of users with different capabilities. For example, the Blackboard system can be used by students, instructors, graders, visitors, and administrators. Students can view the course materials, post discussion threads, and submit assignments. Instructors can post course materials and grade assignments. Graders can grade assignments. Admins can manage users and customize many configurations. The ProjectPortal system may also have different end-user roles, such as students, faculty members, and administrators. Students will be able to add, delete, and edit their own projects. Faculty members will also be able to add their students' projects and provide project feedback. Admins will be able to customize settings. Any user will be able to search the projects without logging in. This is particularly beneficial to prospective students.

Therefore, at the initial planning phase of the project, we should try to list all possible end-user roles and then elicit the possible features or capabilities needed for each.

The INVEST Principle

The INVEST principle was created by Bill Wake to describe the criteria for good user stories. INVEST stands for independent, negotiable, valuable, estimable, sized appropriately or small, and testable. These criteria are explained below:

- **Independent**—The user story should be self-contained, with no inherent dependency on another user story. For example, “Add a Project” should not depend on other user stories such as “List All Projects.” However, be aware that the tasks to implement or test user stories may depend on each other. For example, if the “List All Projects” user story is already implemented, we can test the “Add a Project” by checking whether the newly added story is listed.
- **Negotiable**—User stories can be changed and rewritten until they are parts of an iteration (such as a sprint) and ready for implementation.
- **Valuable**—A user story must deliver value to the end user. That is the reason that the template has the “so that…” part.
- **Estimable**—The developers should be able to estimate a user story’s complexity and the time to complete it. This is usually done by analyzing the tasks involved in implementing the user story and then estimating the complexity and duration of each task. Agile uses the point scheme to estimate the relative complexity of user stories.
- **Sized appropriately, or small**—User stories should not be so big as to become impossible to plan/task/prioritize with a degree of certainty. If a user story is too big, it should be split into multiple user stories. A user story should be able to fit in one sprint.
- **Testable**—The user story or its related description must provide the necessary information to make test development possible. This is very important. In general, the acceptance criteria or tests should be provided for each user story.

Conversation and Acceptance Tests

While a short description can clearly express the customer’s intentions, there is no way that a short sentence can provide all the details needed by developers to correctly implement this user story. The user story is starting point of the conversation between the developer team and the customers about all necessary details. The conversation may be only direct oral communication, or it may be written down on Post-it Notes that are attached to the story index card. Such conversations are very important for developers and customers to correctly understand the requirements and have a consensus about what should be done for each. For example, the following can be written down on a Post-it Note for the “Add a Project” user story.

Add a Project

As a project owner, I want to add my project into the system so that other users can view it.

Conversation for the “Add a Project” User Story

- The system should have an “Add Project” button.

- The user should already be logged in.
- Project information includes the project title, project description, keywords, authors, and possibly various links.
- Project title, description, authors, and keywords are mandatory, and the responses in these fields should only contain valid characters, such as letters, numbers, spaces, and hyphens.
- Project links should be in the valid format.

While conversation can be informal and oral, the developers need to have some way to confirm that the user story is implemented correctly.

Acceptance tests serve this purpose. It is VERY IMPORTANT to have acceptance tests in place before the user stories are implemented, so that the developers can see that their implementation will be accepted if they follow that criteria. Acceptance tests should also be written with or by customers as part of the requirement-analysis activity, as confirmation of a user story. Customers should be able to test the system easily by following those acceptance tests. For example…:

Confirmation (Acceptance Criteria)

- User is already logged in.
- Click the “Add Project” button.
- Check that the title, description, authors, and keywords fields are not empty.
- Check that the title, description, keywords, and authors fields contain only valid characters.
- Check that the project is added to the “My Projects” list.

To clearly describe the testing setup, input, and output, the following template is commonly used to write acceptance tests:

Given…(setup/preconditions),
When…(input/actions)
Then…(expected outputs)

Tools such as JBehave, RSpec, and Cucumber encourage use of this template, though it can also be used purely as a heuristic, irrespective of any tool.

For example…:

Given that I already logged in, and the “Add Project” button is available,
When I click the “Add Project” button,
Then the “Add Project” screen is displayed, with the following fields available: title, description, keywords, authors, and links.

Given that the “Add Project” screen is displayed,
When I input a valid project title (e.g., “ProjectPortal”); description (e.g., “ProjectPortal is a web-based application in which students and faculty members can publish their completed and ongoing project information to share with

others."); authors (e.g., "Yuting Zhang"); and keywords ("software engineering," "project-based learning," "web application"); and click the "Submit" button,
Then the new project is added to the "My Projects" list displayed on the screen.

Given that the "Add Project" screen is displayed,
When I click the "Submit" button without providing any valid project title or description,
Then an error message is displayed: "Please input a valid project title and description."

Acceptance tests are not repetitions of the user story, but confirmations. They should provide enough details for the testers to easily verify whether the user story has been implemented correctly. A common mistake that students usually make is entering a vague description of the expected output or simply repeating the user story. A *really BAD* acceptance test is shown below:

Given that I already logged in, *(Good precondition; any additional precondition or setup needed?)*
When I want to add a project, *(How to add a project is not clear at all.)*
Then the project is added. *(How should the user know the project is added successfully?)*

A single user story should have multiple acceptance tests to verify different details and test different cases—not only the successful case, but also the failure case. In the above example, we show the failure case, when the project title and description are invalid. Make sure that your acceptance tests address all cases.

Tasks and Estimation

After specifying all necessary acceptance tests for a user story, developers need to analyze the story to understand what tasks are involved in its implementation. All possible tasks should be detailed, and each task should be estimated and assigned. Estimation is usually in person-days or person-hours. If it is hard to give a specific number, a range can be used in the estimation. Each task should also be assigned to members of the developer team to ensure that it is completable. A developer may be responsible for tasks from multiple user stories, not just one. For example, the tasks for the "Add a Project" story may include the following:

- Create a project class (YZ)—0.5 person-hours
- Create UI mock-ups for managing project creation, deletion, editing (JD)—0.5 person-hours
- Implement UI to add (create) a project (JD)—2.0 to 5.0 person-hours
- Implement the database to perform CUID on projects (YZ)—2.0 to 5.0 person-hours
- Integrate all components and testing (YZ)—2.0 to 5.0 person-hours

Here, the initials of the responsible developers (YZ and JD) are specified for each task, as well as the estimated person-hours.

When a Story Is Ready

A user story needs to be ready before the developer team starts to implement it. Kenneth Rubin (2012) lists the following criteria:

- Business value is clearly articulated: Every user story should specify the business value. The “so that…” part of the template helps the team reflect on and justify the business value of that user story.
- Details are sufficiently understood by the developers so that they can make an informed decision as to whether the tasks can be completed. When performing the task analysis for each user story, make sure not only to list the tasks, but also to figure out how each task can be carried out. An additional feasibility study or risk analysis may be needed.
- Dependencies are identified, and no external dependencies would block the user story from being completed.
- The team is staffed appropriately to complete it. Make sure that all task assignments are proper, and there is accountability for the story.
- The user story is estimated and small enough to be completed comfortably in one sprint.
- Acceptance criteria are clear and testable.
- Performance criteria, if any, are defined and testable.
- The team understands how to demonstrate the results at the review.

When a Story Is Done

A story is not done unless the following criteria are met (Rubin, 2012):

- Design reviewed
- Code completed
- Refactored, in standard format, commented on, checked in, inspected/reviewed
- Tested
- Unit, integration, regression, platform, and language tested
- Zero known defects
- Acceptance tested
- End-user documentation updated
- Live on production servers

User-Story Management

Product Backlog

Used in the Scrum, the product backlog holds a list of user stories that need to be done in the current projects. In the planning phase, the team uses the product backlog to describe the high-level requirements of the software.

The stories in the product backlog can be prioritized, deleted, and modified. Additional user stories can also be added to the product backlog. All user stories in the product backlog should have business value and be prioritized. User stories with top priority are for the current iteration (or sprint). Each story for the current sprint should be well analyzed, with all tasks and acceptance tests specified.

The items in the product backlog are called PBIs (product-backlog items). Different levels of detail are required for PBIs based on their status. Initially, PBIs are high-level ideas, ready for consideration to be included in the relevant product. The important PBIs are put at the top of the list, ready for refinement. When all necessary details are acquired, they are ready for implementation.

Velocity and Story Points

While it is hard to estimate the absolute effort (in terms of person-hours) of some user stories, it is much easier and more feasible to compare efforts between different user stories. Story points are used in Agile to reflect the relative effort associated with the story from the developer's perspective. A 2-point user story is definitely more complex than a 1-point user story and may need double the effort.

Different scale schemes may be used in the point assignment. A commonly used scale scheme is the linear scale, which assigns points such as 1, 2, 3, and 4 to different sizes of user stories. An exponential scheme assigns points such as 1, 2, 4, and 8 to address exponentially increasing complexity of user stories. Even the Fibonacci scale is sometimes used to assign points such as 1, 2, 3, 5, and 8 to stories. Understand that the user-story points reflect the relative complexity or sizes of user stories in a project, not the absolute sizes. Each project team has their own understanding of what a point means, and it differs from project to project. For example, in one project, a 1-point user story may need one person-day's worth of effort, roughly. In another project, 1 point may only need one person-hour's worth of effort.

Velocity is the number of story points completed in one iteration. Velocity helps us measure the project's progress and team productivity, plan future iterations, and adjust estimates of points for each user story. In particular, historical velocity data can help us estimate and plan new iterations. While it fluctuates from iteration to iteration, it should be quite stable over the long term. Introducing new tools, getting training, and changing the process may cause temporary drops in velocity, but may improve the velocity in the long run. Working overtime may increase the velocity temporarily, but is not a sustainable approach and may harm productivity in the long run.

Epics and Themes

Epics are user stories that are too big to implement in a single iteration and therefore need to be disaggregated into smaller user stories at some point. There are different techniques to splitting big stories. This article, Patterns for Splitting User Stories lists a number of different techniques for splitting user stories based on work-flow steps, business-rule variations, degree of effort required, simple versus complex, variations in data, different data-entry methods or operations, etc.

A theme is a collection of related user stories. For example, we may have several themes in the ProjectPortal project: user management (e.g., user registration, login, logout); project modification (i.e., add, delete, edit project); search project; and feedback (e.g., comment, rating, feedback). Themes are often used to organize stories into releases or so that various subteams can work on them.

Planning

While Agile values "responding to change over following a plan," that doesn't mean that planning is not important. Rather, planning is a very important part of the software-development process. Instead of big, up-front planning, adaptive planning is needed throughout the project. This includes planning up-front and for each iteration (sprint planning).

Initially, we need to plan for the entire project. A very detailed plan is usually hard to follow as the project is executed and thus unnecessary. However, we still need a plan of coarse granularity. At this stage, we should decide the project's scope and roughly estimate its time and cost. We want to set some milestones and iteration goals. For our project, the deliverable of this initial planning is the SPPP (software-project proposal and planning) document. In this document, you should clearly describe the project's scope and high-level requirements. To make it more adaptable, split high-level requirements into three categories: essential, desirable, and optional. Essential requirements are those that can be implemented within a semester. All essential requirements should be added into the product backlog. The desirable and optional features can be added later. A rough estimation of each essential requirement is needed to ensure that it can be implemented and tested before the end of the semester. Don't be too optimistic, and leave some room for unexpected situations.

If a requirement (or a user story) is estimated to be too big (e.g., requiring more than 50 person-hours), you may consider breaking it into smaller ones. Take account of all possible, related tasks, including learning, analysis, refinement, design, debugging, testing, and communication. Use the smallest, simplest story as the 1-point story. Define points for other stories based on that.

The estimation of a user story can be quite different under various assumptions. For example, if the database schema and connections are already implemented, it will take much less time to implement the "Add a Project" user story. Usually, the first several stories need much more time, as you will need to complete a number of tasks that are common to the other stories—such as database design, setup implementation, and framework testing.

After the estimation and based on customers' preferences, we can arrange these requirements into different iterations and set the iteration goals and milestones. While we should try to stay aligned with this initial rough plan as much as possible, it is still possible to modify it later, particularly given new customer requests. In general, we should try to avoid big changes (e.g., adding or deleting essential requirements), but always be ready to adjust and refine the original plan (e.g., by adding or modifying details of requirements or moving requirements from iteration to iteration).

We should also start each iteration with planning, which includes setting the iteration goals, selecting the PBIs for the current iteration, preparing the resources, and assigning tasks. In general, essential requirements should

have higher priority than desirable and optional ones unless it is very easy to implement a desirable or optional feature along with some essential ones.

Scope Creep

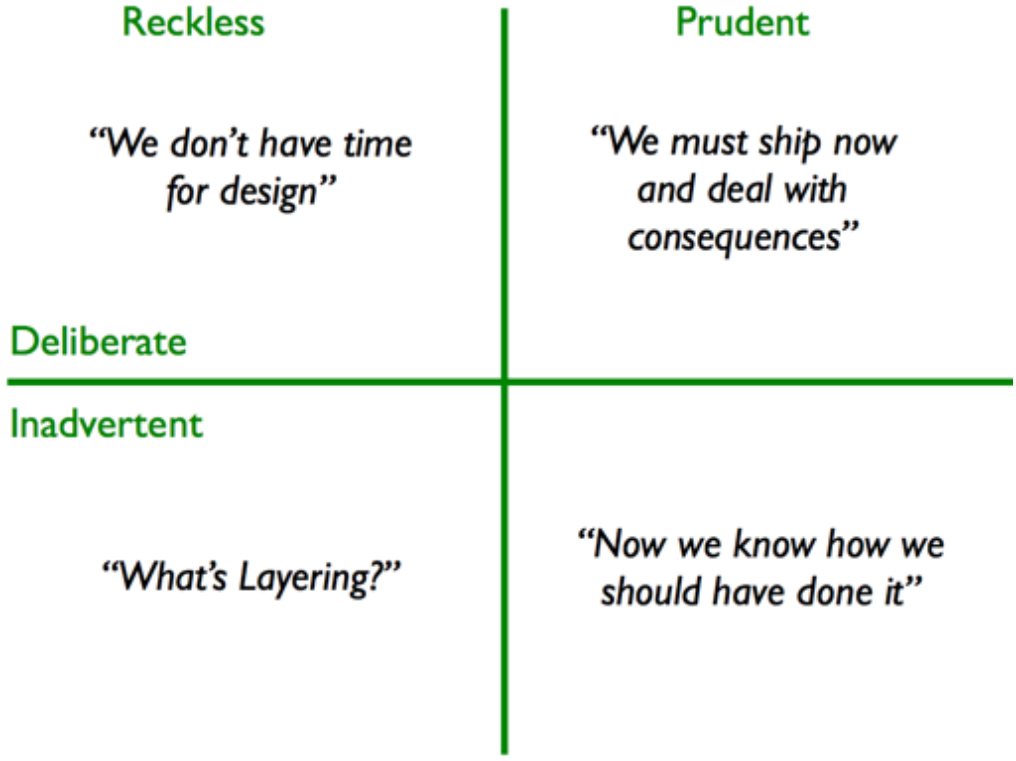
While changeability is an inherent feature of the software, and Agile welcomes changes, changes need to be carefully managed to avoid the adverse effect. One such effect is scope creep. It occurs when the scope, deliverables, or features of a project expand from what was originally set—without being accounted for with additional time or budget. The consequence will be increased cost, missed deadlines, or reduced quality. To avoid the scope creep, it is important to do the following:

- Have a clear vision of the project .
- Have a realistic estimation.
- Manage the change.
- Communicate clearly and often.

Manage Technical Debt

The term “technical debt” was first written by Ward Cunningham. Nowadays, it refers to shortcuts that are purposely taken or problems such as bad design, defects, insufficient test coverage, and excessive manual testing.

Martin Fowler presents the technical-debt quadrant in the following diagram.



The Technical Debt Quadrant

Some technical debt, such as reckless debt, can be eliminated through proper planning; other debt may be unavoidable. For example, we cannot perfectly predict up-front how the system works; therefore, our initial design may need to change.

Just like financial debt, technical debt requires interest payments, in the form of extra development effort. We can choose to continue paying the interest or pay down the principal. In any case, the debt needs to be managed explicitly to avoid letting the accumulation exceed what we can take and thus cause severe consequences—such as increased time to delivery, increased development and support costs, decreased customer satisfaction, and a significant number of defects.

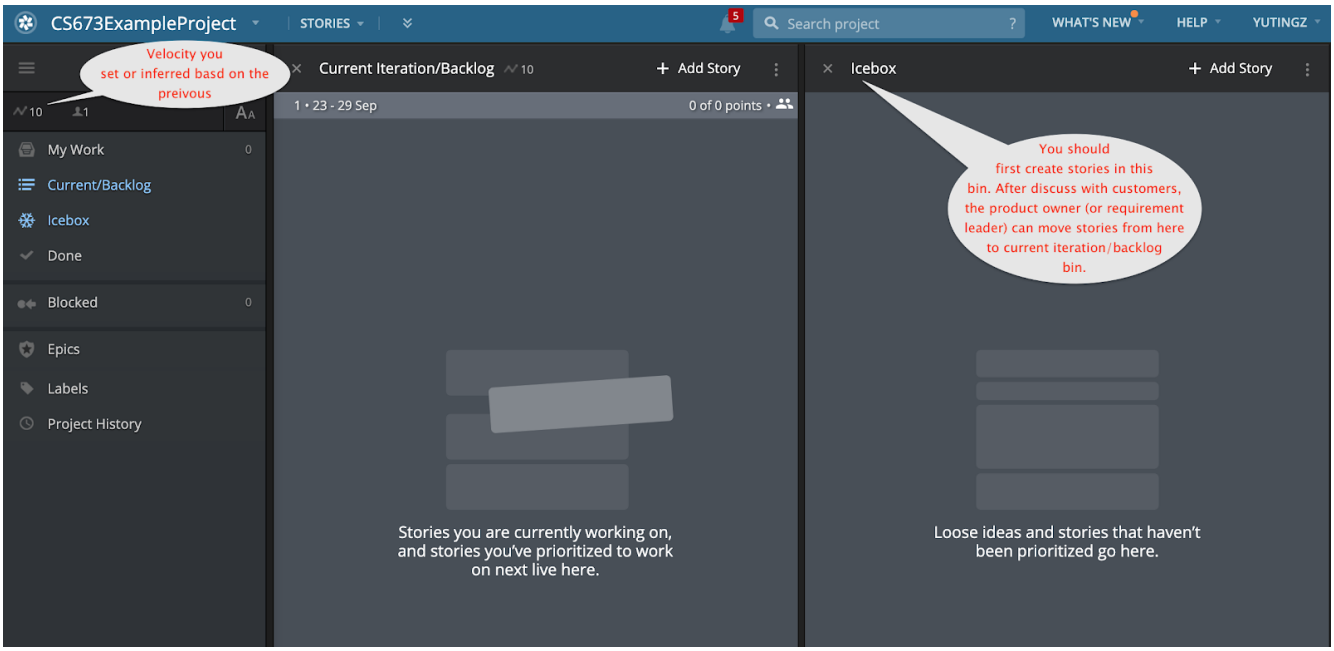
There are different ways to manage technical debt (Rubin, 2012):

- Manage accrual of technical debt
    - Use good technical practices, such as test-driven development (TDD) and continuous integration
    - Use a strong definition of “done”
    - Properly understand technical-debt economics
- Make it visible
    - At the business level, for example, by connecting it with financial debt and tracking velocity over time
    - At the technical level, by explicitly tracking it using defect-tracking system or separate technical-debt backlog, or simply in product backlog
- Repay the debts
    - Repay high-interest debt first
    - Repay some incrementally
    - Repay some while performing custom-valuable work
    - Be aware that not all debt needs to be repaid—for example, if developing a throwaway prototype

Use an Online Tool to Manage User Stories

Instead of writing down user stories on index cards and Post-it Notes, we can use many online tools that help manage user stories and requirements. This is particularly useful for distributed teams. One such tool is Pivotal Tracker, which we will use in our group project. Pivotal Tracker is a simple tool designed particularly to manage requirements written as user stories.

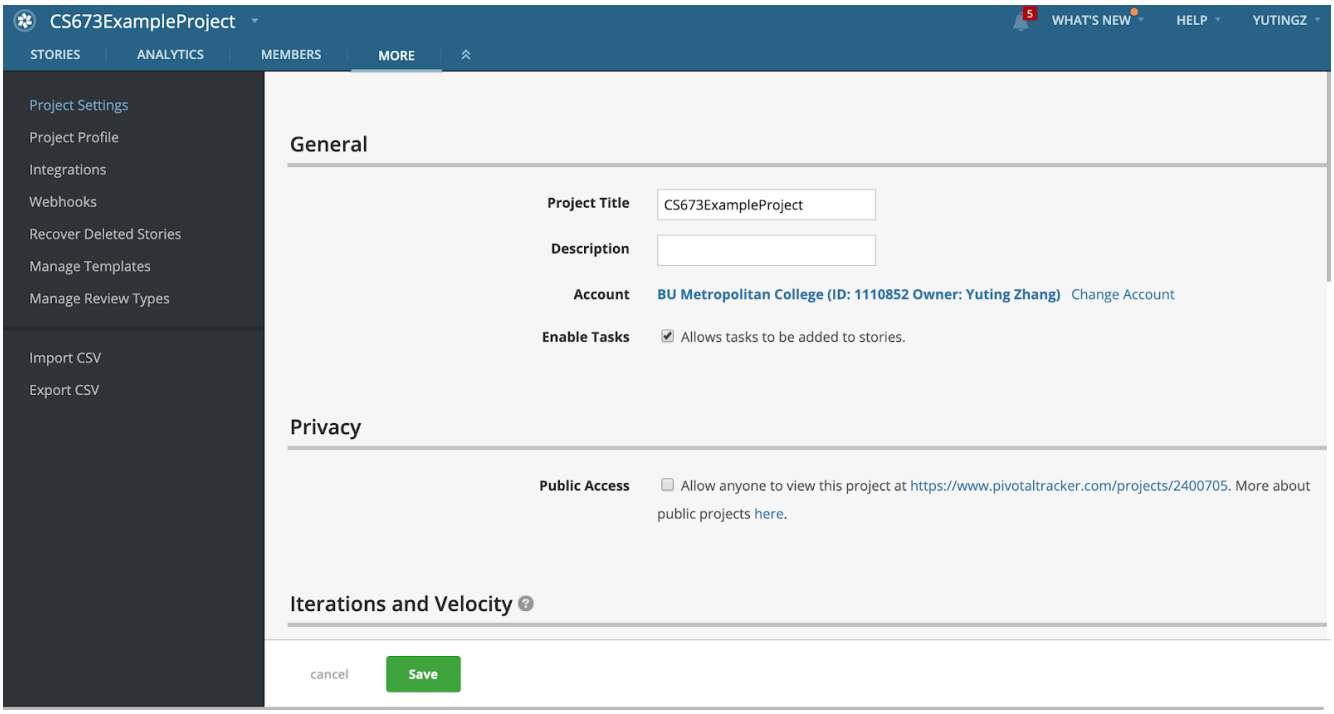
The following screenshots show the interface of Pivotal Tracker.

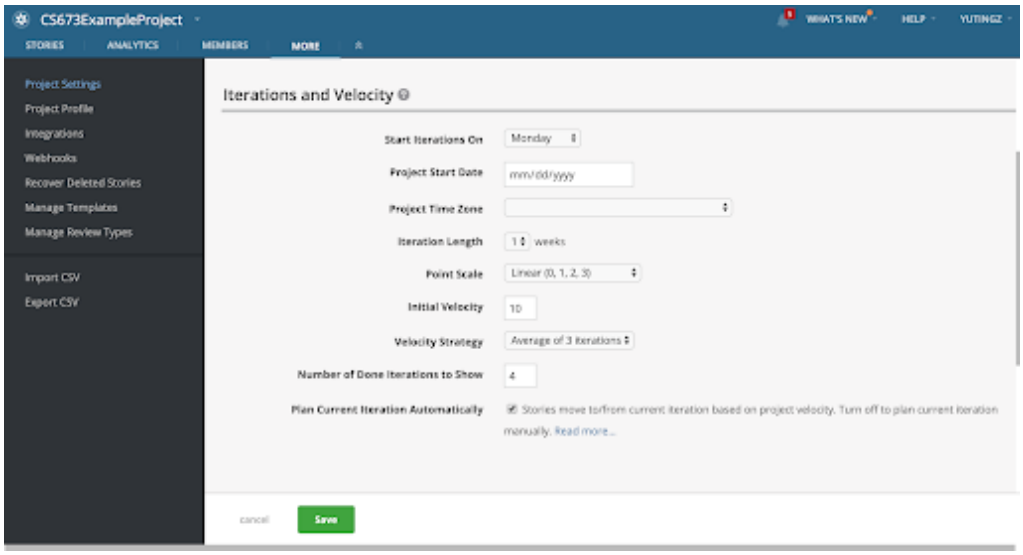


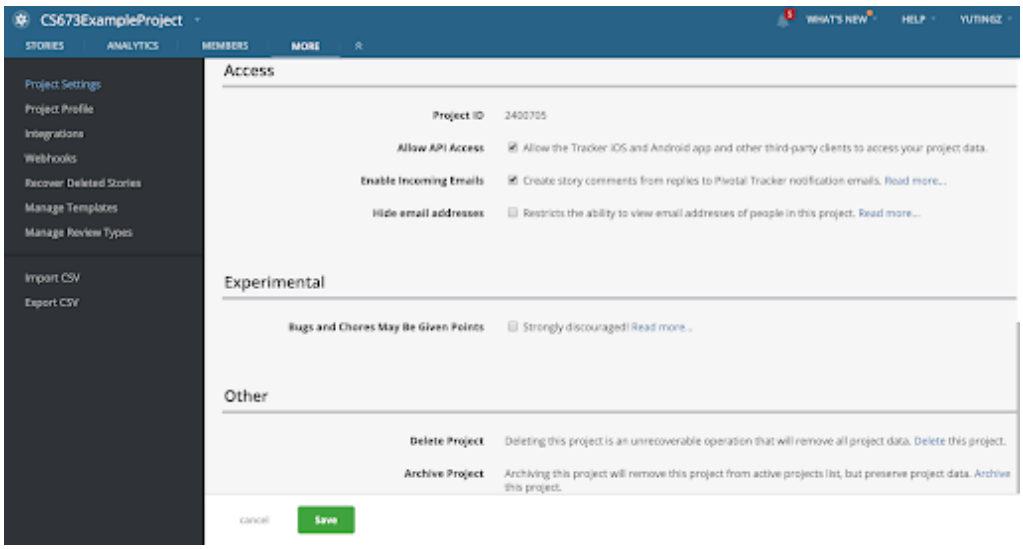
Pivotal Tracker UI

Project Settings

When clicking the menu item “MORE,” you will find a number of features for customizing the project on Pivotal Tracker. You can set general project information, such as title and description. You can set the iteration length to one, two, three, or four weeks. There are several point-scale options: linear, exponential, and Fibonacci. You can also set how the velocity is calculated and whether the current iteration is automatically planned. If it is set, then the first several user stories in the current iteration and backlog, with the total story points as the velocity, will be moved into the current iteration. For example, if the velocity is set to 10, and each user story in the current iteration and backlog bin is 3 points, then the first three user stories will be in the current iteration, as their points total 9—the maximum value that is less than the velocity of 10. Another setting to pay attention to is “Bugs and Chores May Be Given Points.” In Pivotal Tracker, you can also list bugs and chores as user stories. However, by default, they are not assigned points and thus will not affect velocity, because these items are for the developer team, not for customers. if your team wants, you can also assign points to bugs and chores, though this practice is generally discouraged.

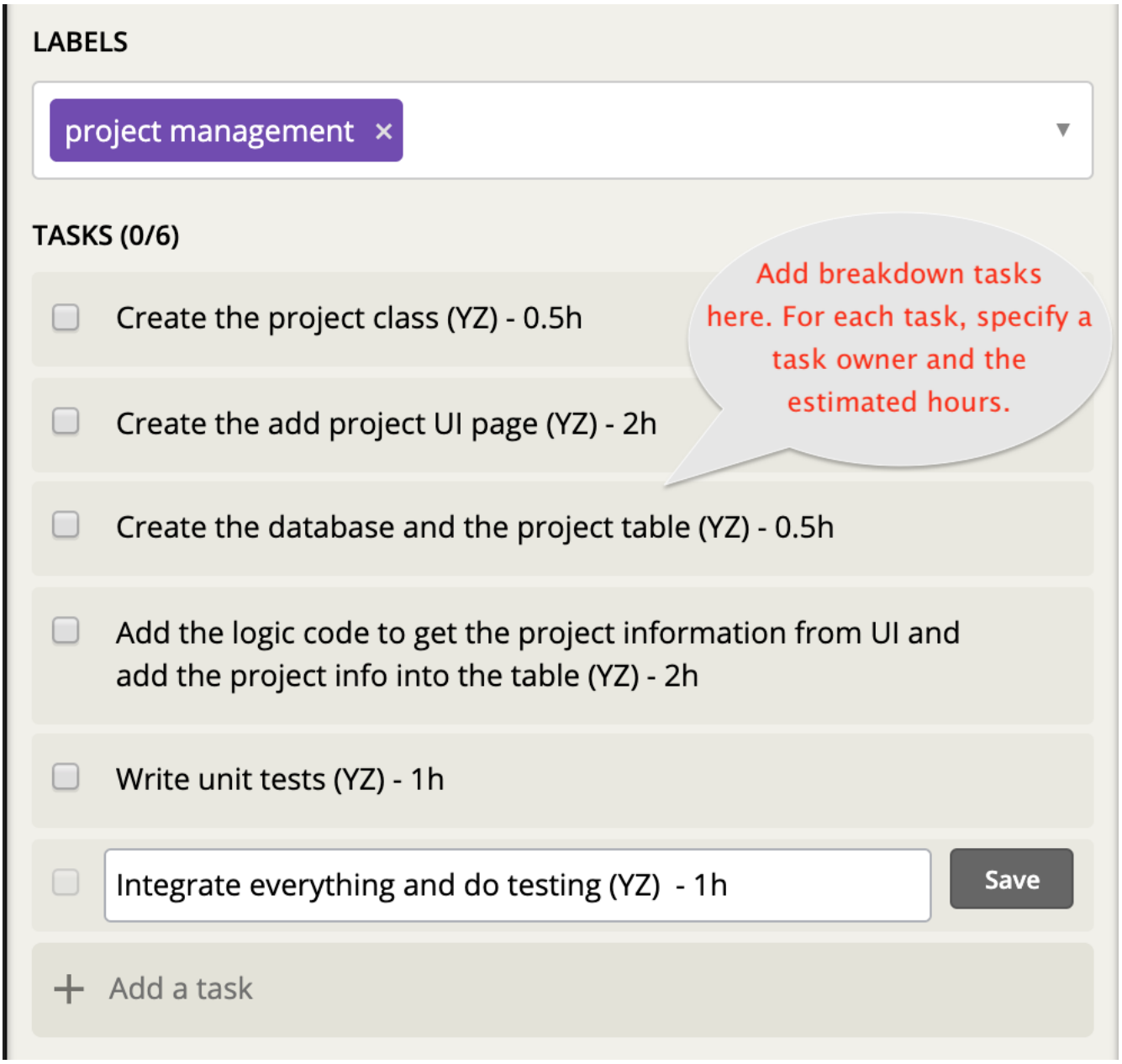


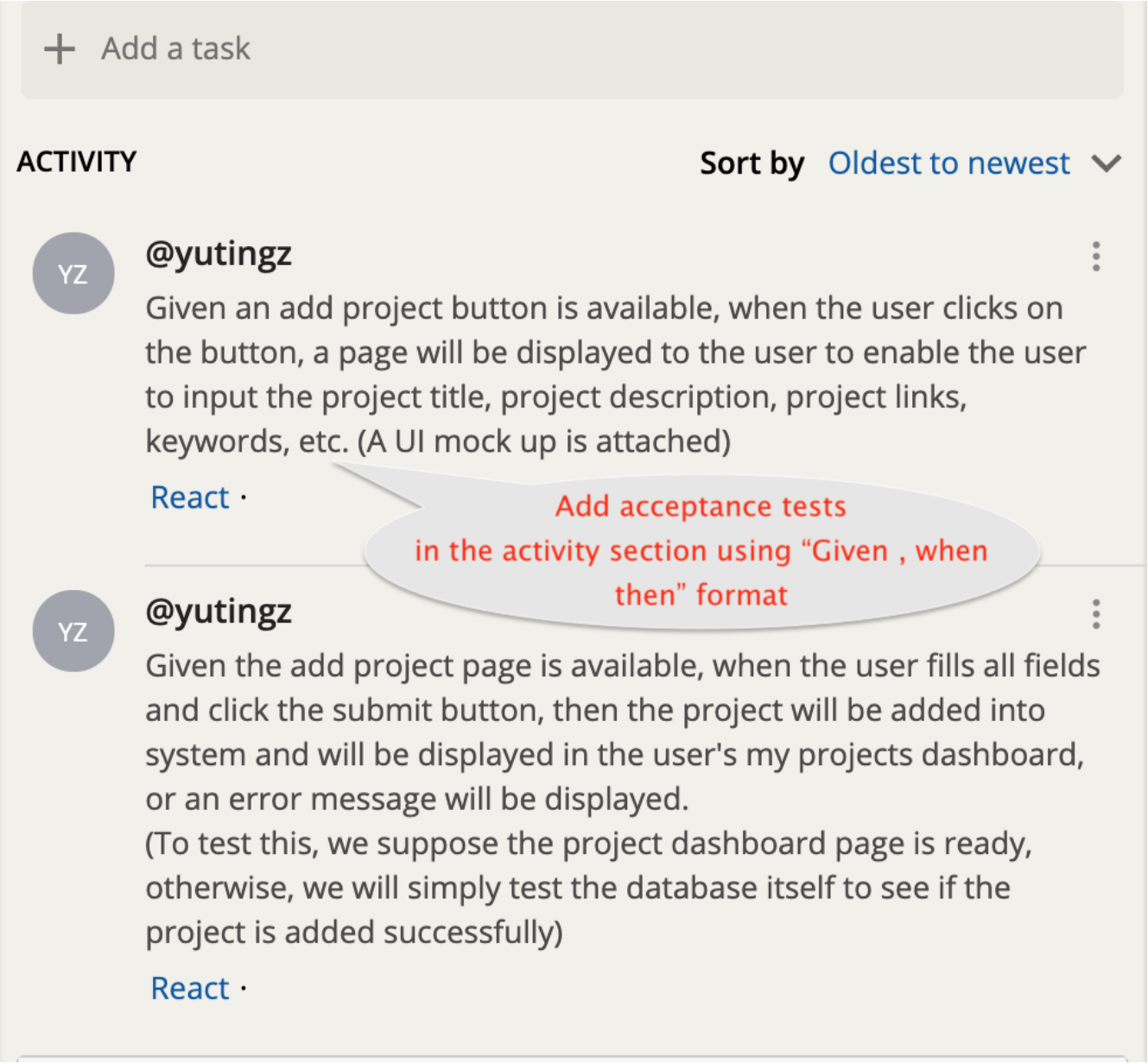




Add User Stories

When you click “Add Story,” the following page will be displayed to let you add a user story. The following example shows the “Add a Project” story for the ProjectPortal project. You can add labels for each user story for easier indexing and grouping different user stories together. For each story in the current iteration, the tasks should be analyzed in the iteration-planning phase. You should also assign the task executors and estimate the time for each task. In addition, conversations about the user story can be added in the activity session, together with acceptance tests.


LABELS
project management ×
TASKS (0/6)
Add breakdown tasks here. For each task, specify a task owner and the estimated hours.
Create the project class (YZ) - 0.5h
Create the add project UI page (YZ) - 2h
Create the database and the project table (YZ) - 0.5h
Add the logic code to get the project information from UI and add the project info into the table (YZ) - 2h
Write unit tests (YZ) - 1h
Integrate everything and do testing (YZ) - 1h
Save
Add a task



User-Story Exercises

Consider the Blackboard system, which we are all familiar with. Please add a user story in the Icebox of a sample project in the Pivotal Tracker using the following URL: Pivotal Tracker. You should add a task breakdown, estimations, and at least three acceptance tests there.

Nonfunctional Requirements

So far, we have mainly focused on functional requirements (also called features), which define the services that the software system can provide, usually phrased as actions. There is another type of requirement: nonfunctional

requirements, which define the quality and constraints of the services provided, usually phrased as nouns or assertions.

There are several types of nonfunctional requirements (NFRs):

- **Quality attributes ("-abilities")**—Usability, reliability, availability, maintainability, portability, performance, security, etc. When defining NFRs in this category, it is not enough to simply use these "-ability" words; we need to provide a quantifiable and testable description. For example, instead of simply stating, "The system is scalable," we need to be specific and explain the scalability in a testable way, such as, "The system shall support 10,000 concurrent users."
- **Constraints**—Platforms, development media, operations, packages, licenses, regulations, etc.
- **External interfaces**—Hardware, other software, communication with external agents.
- **Error handling**—Ignore, warn user, allow unlimited retries, log and proceed anyway, substitute default values, shut down.

Both functional and nonfunctional requirements need to be specific, quantifiable and testable. They should also be consistent, feasible, valuable, and traceable.

Ambiguous, Not Testable, Bad	Specific, Quantifiable, Testable, Good
• The system is available most of the time. • The system is scalable. • The system is user friendly. • The system is fast. • The system is secure.	• The system shall support "five nines" availability • The system shall support 10,000 simultaneous requests. • AES-256 is used to encrypt all personal data. • The application must be executable on any 1GH Linux computer. • The application must interface with a model of 1234 bar-code reader.

Nonfunctional requirements are not individual features, so they are not PBIs (product-backlog items). Instead, they are qualities and constraints that span the PBIs. They are usually described in the acceptance tests and need to be tested at the system level.

User stories are mostly used to represent high-level functional requirements. Sometimes people also use them to represent nonfunctional requirements. For example, Mike Cohn defines the following nonfunctional requirements as user stories:

- As a customer, I want to be able to run your product on all versions of Windows from Windows 7 upward.
- As a user, I want the site to be available 99.999% of the time I try to access it so that I don't get frustrated and find another site to use.
- As someone who speaks a Latin-based language, I might want to run your software someday.
- As a user, I want the driving directions to be the best possible 90% of the time and reasonable 99% of the time.

However, Cohn did caution that the user-story template should only be used as a thinking tool. It should not be used as a fixed template for all nonfunctional requirements.

Test Yourself

Which of the following are good functional requirements?

The application should be user friendly.

The application should be scalable.

The application should use an SQL database.

Users should be able to log into the application to authenticate themselves.

Test Yourself

Which of the following requirements are nonfunctional?

All personal data should be encrypted using AES-256.

The password for login should be longer than eight characters.

Students can submit assignments through the Blackboard system.

Files attached to assignment submissions should not exceed 1 MB each.

Test Yourself

What does “INVEST” stand for?

Independent, negotiable, valuable, estimable, sized appropriately or small, and testable.

Test Yourself

Write an acceptance test for the “Submit an Assignment” user story for the Blackboard system.

Test Yourself

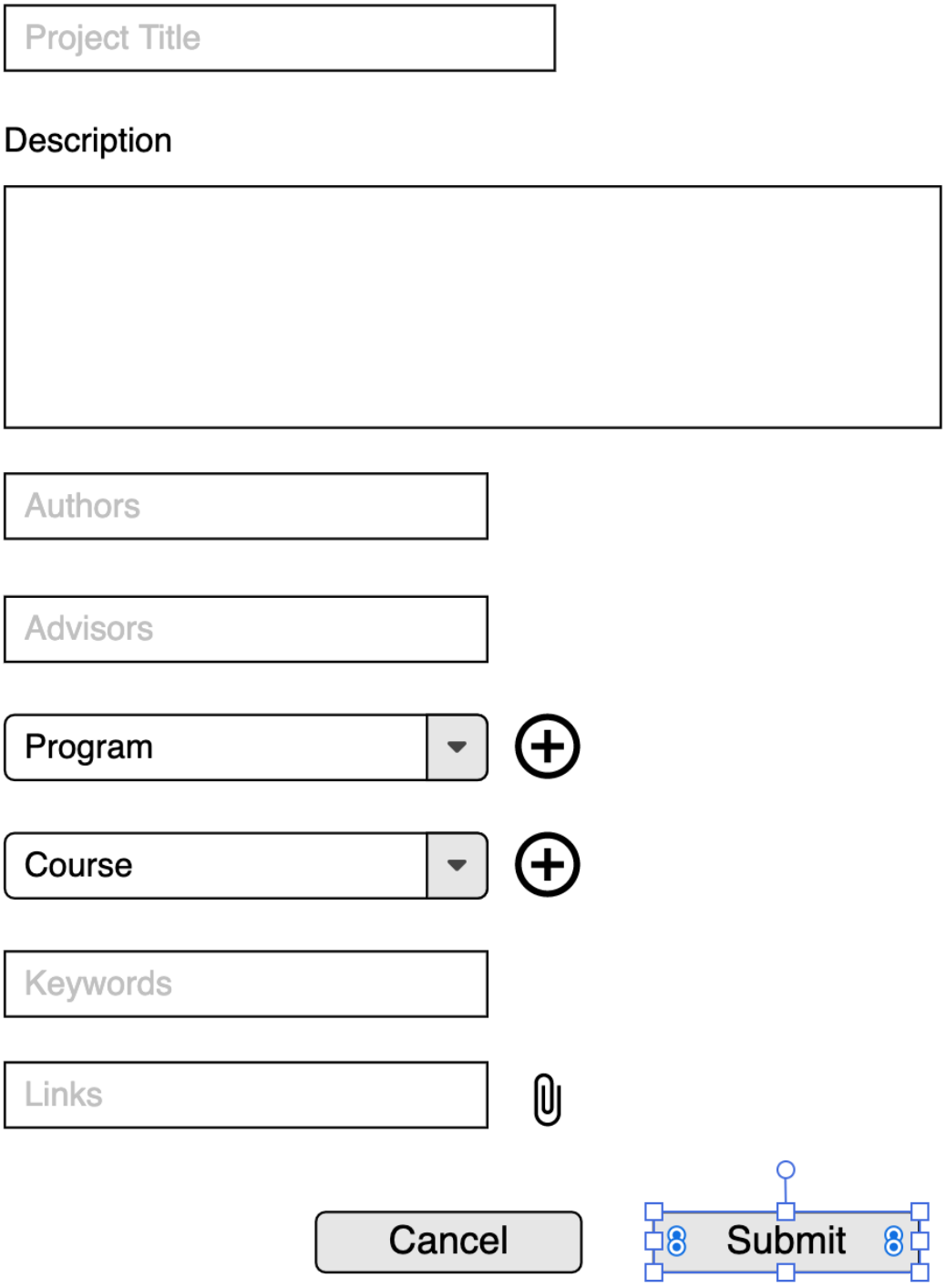
List the criteria for “ready” and “done,” respectively, for a user story. (Check the lecture notes.)

Topic 2: From Requirements to Design - UML Class Diagrams and State-Transition Diagrams

There are other ways besides user stories to represent requirements. Use-case models are used in the unified process and have been used widely in the last decade. We will discuss these in our later modules.

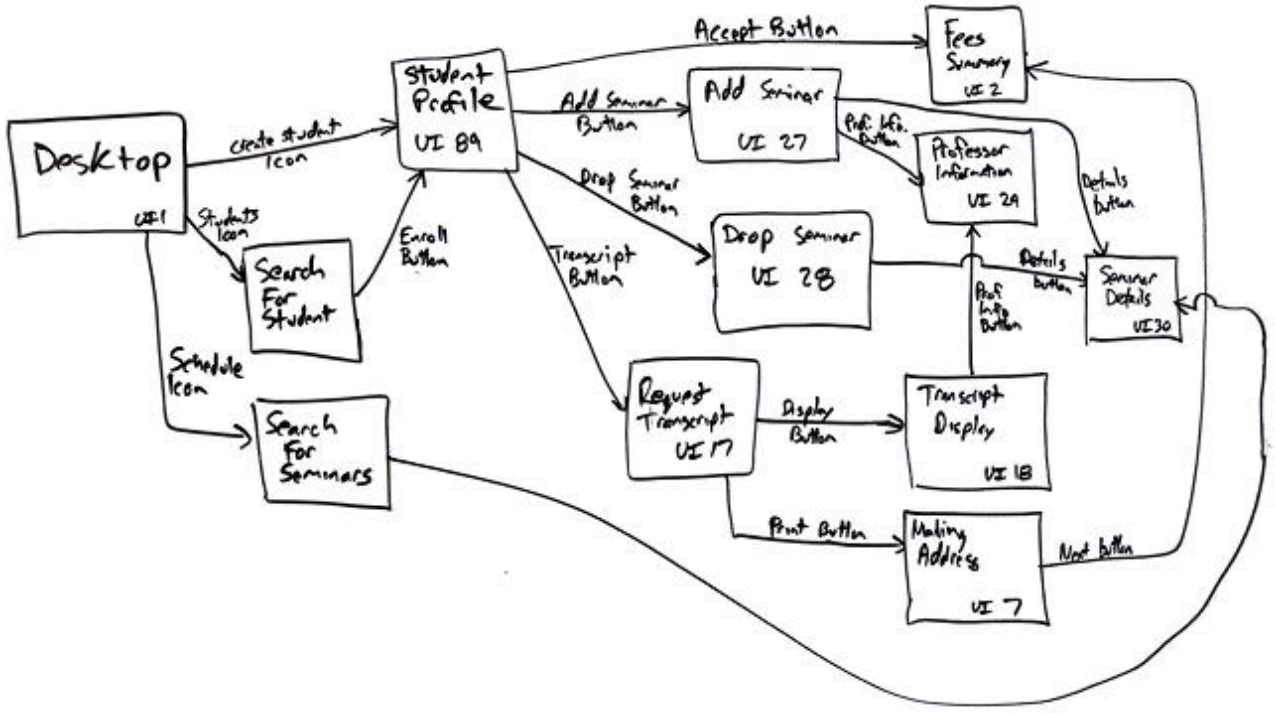
UI Mock-Ups and State-Transition Diagram

To visualize requirements, low-fidelity (lo-fi) UI mock-up screens are often accompanied by user stories or use cases to provide customers with better ideas about the software system to be implemented. The easiest approach is to hand-draw the UI mock-ups using pen and paper, maybe on Post-it Notes that are attached to the user stories. There are also a lot of tools that help create prettier mock-ups, such as Balsamiq, Mockplus, wireframe.cc, and Moqups. In the requirement-analysis phase, the focus is to convey the basic idea about the UI without many details, such as color, fonts, and pixels. It is a fast, cheap way to brainstorm ideas and get early feedback. For example, the following diagram shows an example of the “Add a Project” mock-up screen:



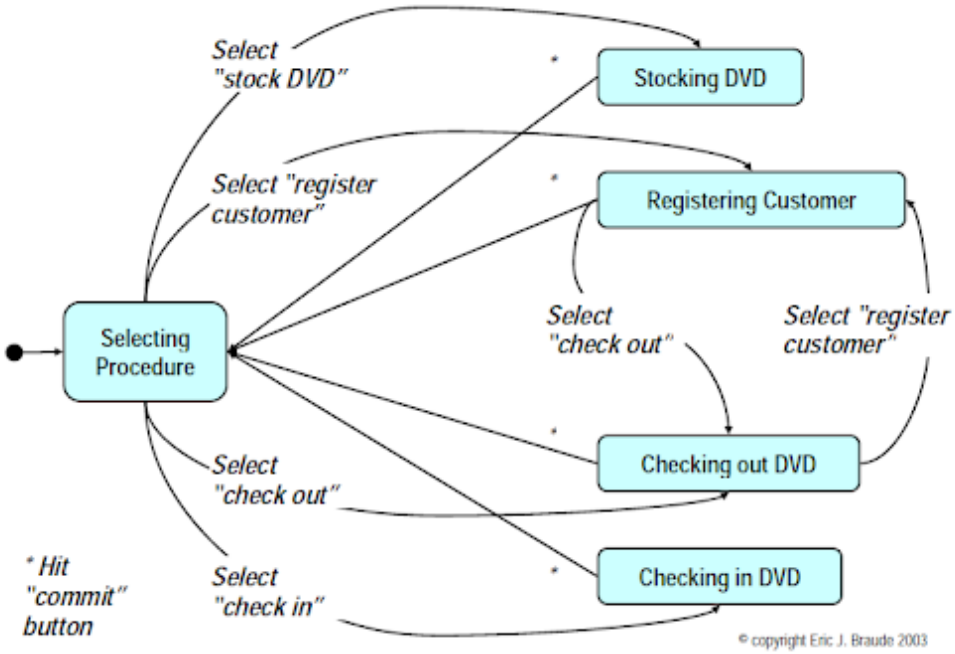
"Add a Project" Mock-up

To understand the navigation among UI screens and thus the relationships between requirements, UI flow diagrams (or UI storyboards) can be used to show the transitions between different UIs. The following diagrams shows an example of the flow of UI.



UI Flow Diagram

UML state-transition diagrams are also used to model the UI screen transition. For example, each state is a name of screen, and the transition line indicates the events or actions that trigger the navigation from one screen to another. The initial state is represented by a solid circle. The following diagram shows an example from Braude's books.



GUI Transitions for Video Store

Entity-Boundary-Control (EBC)

Nowadays, object-oriented design and languages are commonly used in software engineering. To bridge the gap between high-level requirements and design, we should first identify application-domain classes/objects. There are usually three types of domain classes/objects:

- **Entity**—Represents the persistent information tracked by the application
- **Boundary**—Represents the interfaces between the user and the application
- **Control**—Represents the control tasks (actions) performed by the application

In general, the interface of a system is more likely to change than the control, and the control is more likely to change than entities in the application domain. By separating these three types of objects, the system is more resilient to change.

After understanding these three types, we should identify real-world entities that the system needs to keep track of, then identify interface artifacts, and finally identify the control for coordinating boundary and entity objects. Usually, there is one control object per user story. The main goal here is to find important abstractions in the application domain.

For example, based on the description of the “Add a Project” user story, we can identify the following EBC objects:

- **Entity**—Project, user, the ProjectPortal system,
- **Boundary**—addProject screen (including various text fields, drop boxes, and buttons)
- **Control**—addProject

Test Yourself

CCan you identify entity, boundary, and control classes in a file-browser system such as Windows Explorer or Finder?

- Entities:
- Boundary:
- Control:

Class Diagram

A class diagram is a basic UML diagram to describe objects/classes and their relationships in the system. It can be used in both the requirement-analysis and design phases. It is a structural model used to show the static structure, not the dynamic behavior.

UML (Unified Modeling Language) was created by Ivar Jacobson, Grady Booch, and James Rumbaugh, who also developed the unified process that we introduced in the first module. They are also pioneers of object-

oriented design (OOD). UML is a set of modeling languages using OOD concepts to visualize software systems. It is very useful and widely used. However, we should also be careful not to overuse it. UML 2.0 is a major update. The newest version is UML 2.5.1. More details about UML can be found on its official website: UML.org

UML 2.0 defines 13 diagrams, classified into 3 categories:

- **Structure diagrams**—Class diagram, object diagram, component diagram, composite structure diagram, package diagram, and deployment diagram
- **Behavior diagrams**—Use-case diagram (used by some methodologies during requirements gathering), activity diagram, and state-machine diagram
- **Interaction diagrams**—Sequence diagram, communication diagram, timing diagram, and interaction-overview diagram

The basic component of a class diagram is a class. A class has some attributes and methods. In the object-oriented programming languages like Java, each attribute or method has visibility constraints, being *public(+)*, *private(-)*, or *protected(#)*. Each attribute may specify a type, and each method has its unique signature, including return type, method name, and parameters. The following diagram shows the Project class in the class diagram.

Project
-id: long -title: string -description: string -keyboards: string
+editProject(title: string, description: String):bool +addKeywords(keyword: string): bool

In the requirement-analysis phase, visibility, type, and method signature information are not needed. The phase mainly focuses on the application-domain classes. In the design and implementation phases, additional classes from the framework or design patterns may be added. You can even generate the class diagram automatically from the code, using some tools. For example, both Eclipse and Intellij IDEA have some plug-ins that can generate UML class diagrams from Java code.

A class diagram usually consists of a number of classes that are related to each other. There are several types of class relationships:

- **Inheritance**—This is the strongest relationship. It is usually represented by a line with a triangle head pointing to the superclass. For example, the ComputerScienceStudent class inherits from the Student class. Anywhere Student appears, we can always substitute it with ComputerScienceStudent. The

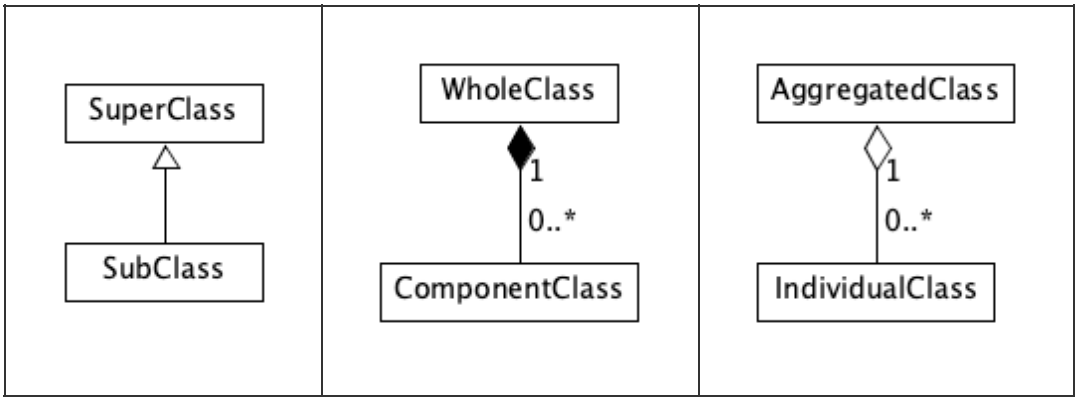
superclass is tightly coupled with the subclass. Any changes in the superclass can affect its subclasses. In Java, both “extends” and “implements” are inheritance relationships.

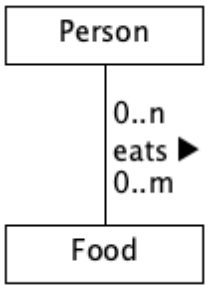
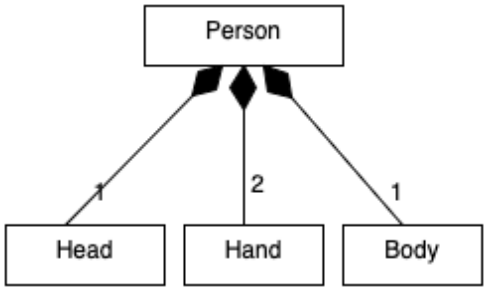
- **Composition**—This is a whole-part relationship. It is represented by a line with a solid diamond head pointing to the whole class. For example, a person has a head and two hands.
- **Aggregation**—This is a loose whole-part relationship. It is similar to the composition relationship, but with a subtle difference. Aggregation implies that the child can exist independently of the parent, while composition implies that the child cannot. For example, we can use the aggregation relationship between a bookstore and a book. It is represented by a line with an empty diamond shape.
- **Association**—Any general association between two classes can be represented simply by a solid line between them. It is also possible to have an association to the class itself. The name of the association can be shown somewhere near the middle of the association line, but not too close to either end of the line. Each end of the line can be decorated with the name of the association end. For example, the association “eats” shows that a person eats some food.

  To be more specific, the multiplicity is specified to describe the numerical relationship. The number annotating each end of a line shows the number of instances of each class. For example, a person can eat 0 to m different units of food. Each food can be eaten by 0 to n people. This shows a many-to-many mapping. One-to-many or many-to-one mapping is also possible in general association. One-to-many mapping is the default in the aggregation and composition relationships. One can use explicit numbers to indicate how many ComponentClass/IndividualClass instances are mapped to one WholeClass/AggregatedClass instance. For example, a person has one head, two hands, and one body.

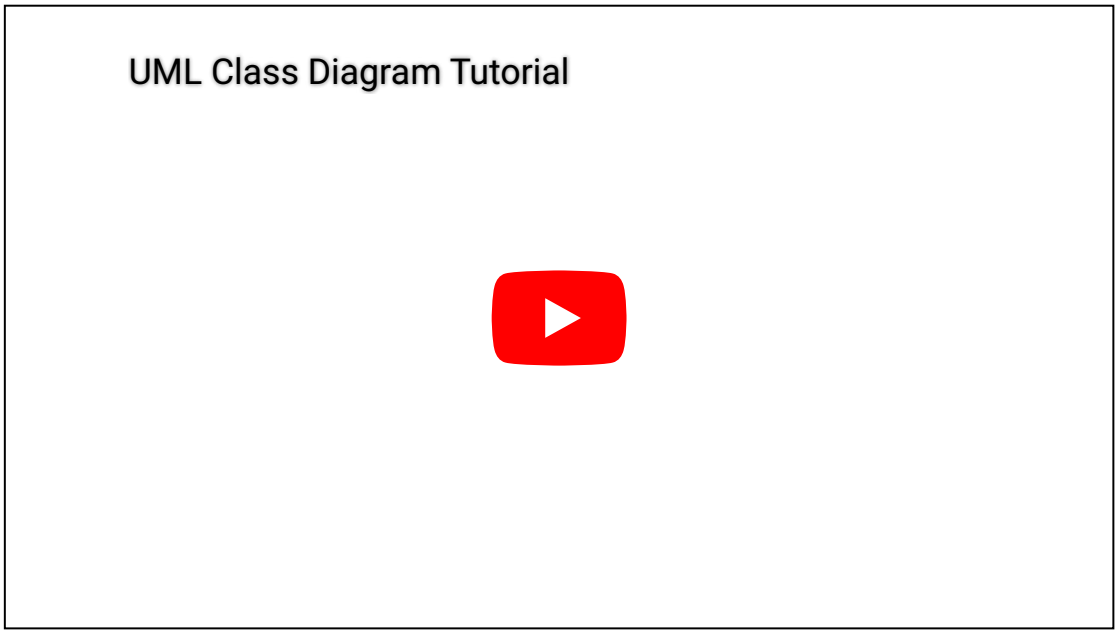
- **Dependency**—This is a much weaker relationship, indicating that one class depends on another class if the independent class is a parameter variable or local variable of a method of the dependent class. It is represented using a dash line with an arrow pointing from the dependent class to its dependency. This relationship is usually not identified in the requirement-analysis phase, but in the design phase.
- **Unidirectional and bidirectional**—By default, relationships are bidirectional, represented by a line without an arrow. However, sometimes we may want to use a unidirectional relationship to indicate that only one class knows the relationship. A unidirectional association is drawn as a solid line with an open arrowhead pointing to the known class. The benefit of using a unidirectional relationship is that only one class needs to be updated for changes. There is no need to worry about consistency. The downside is that you can only access the relationship from one class.

Inheritance	Composition	Aggregation



General Association with Multiplicity	Composition with Multiplicity
	

There are a number of tools that can help you draw UML diagrams, such as UMLet, argoUML, startUML, and Lucidchart. The above diagrams are produced using UMLet. You can also watch the following tutorial for some basics about the class diagram:



How to Create a Class/Object Model

There are four steps to creating a class/object model:

1. Find objects (class)
2. Find the properties (attributes)
3. Find the actions/operations (methods)
4. Find the relationships between classes (e.g.inheritance, aggregation, composition, general association)

Abbott's technique is a simple textual analysis (noun-verb analysis) used to identify possible classes. Nouns are candidates for objects/classes, verbs are candidates for operations/associations, and adjectives are candidates for attributes. For example, suppose the customer provides a scenario of shopping in a toy store, with the following description:

The customer enters the store to buy a toy. It has to be a toy that his daughter likes, and it must cost less than 50 euro. He tries a video game, which uses a data glove and a head-mounted display. He likes it.

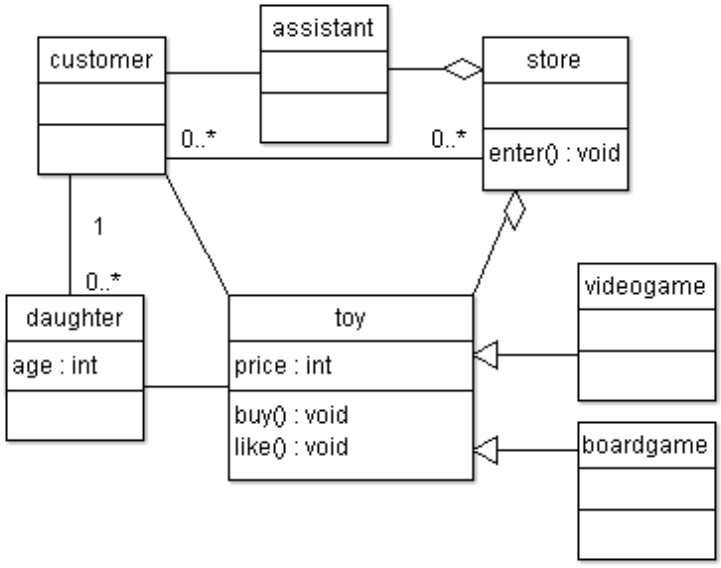
An assistant helps him. The suitability of the game depends on the age of the child. His daughter is only three years old. The assistant recommends another type of toy, namely the board game "Monopoly."

Based on the above description, we can use textual analysis to find nouns, verbs, and adjectives, and identify the following classes:

- Customer/daughter (enter, buy, like, age)
- Store
- Toy, game, video game, board game (use cost, age, dangerous)
- Accessories,
- Assistant (recommend)

The expression *"…is a…"* usually means inheritance, and *"…has a…"* usually means aggregation.

The following class diagram can be used to show these classes and their relationships:



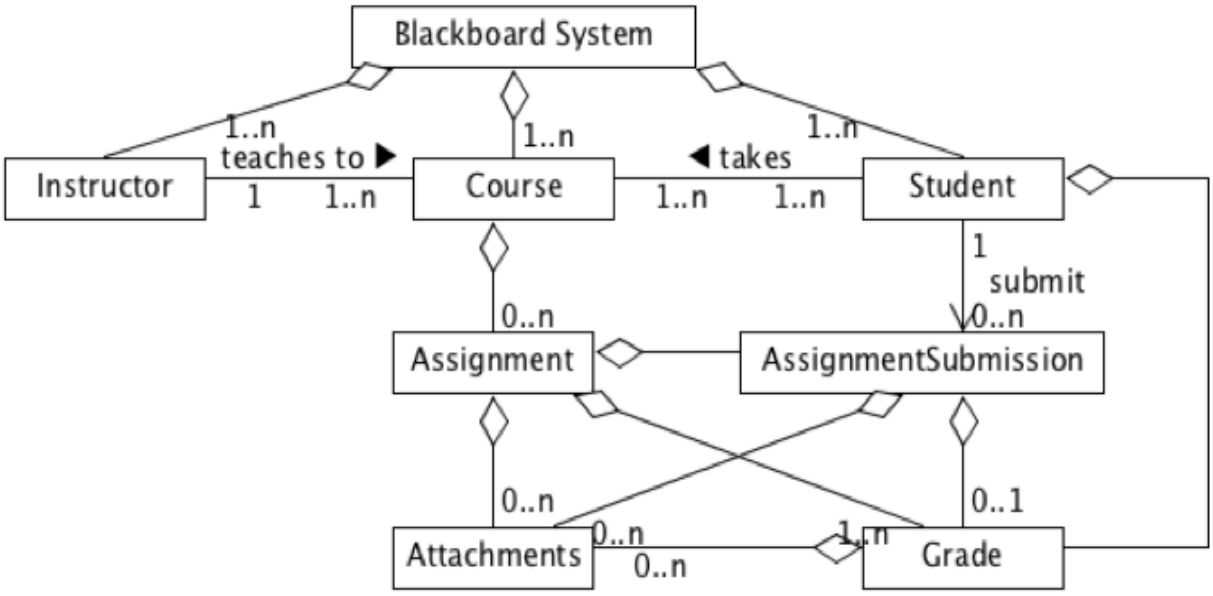
Here is another example. Suppose we have the following description related to assignment submission and grading in the Blackboard system:

> *On the CS 673 Blackboard course site, students can submit their completed assignments by clicking the corresponding assignment. A new page will be displayed. Students can upload their files, write the comments, and click the "Submit" button to submit their assignments. Each assignment has a deadline. After the deadline, the instructor will grade each student's submission by clicking the corresponding submission and assigning a score. The instructor can also add comments or attach additional files as feedback for students.*

Based on the above description, we can identify the following classes:

- **Entity**—BlackboardSystem, Course, Student, Instructor, Assignment, AssignmentSubmission, Attachment, Grade
- **Boundary**—AssignmentSubmissionPage, AssignmentGradingPage
- **Control**—SubmitAssignment, GradeAssignment

We can draw a class diagram to show the relationships between entity classes, as below:



Additionally, we can identify classes based on general world knowledge. Not all classes identified are useful. A list-and-cut process is needed to eliminate any unnecessary ones and leave only a few essential classes. Also, don't forget the aggregated class. For example, in the above "Add a Project" use story, we should also identify the *ProjectPortalSystem* class, in addition to the *Project* class, as well as the aggregation relationship between *ProjectPortalSystem* and *Project*.

Sometimes we may find the wrong abstraction. It is okay; we just need to iterate and revise the model.

Here is the recap of the roadmap to the class model:

- Obtain domain classes and objects from high-level requirements
- Add essential domain classes and inspect the resulting collection of classes
- For each class, specify the required attributes, functionality, and how the class's objects react to events; draft test plans for each; and inspect the results
- Inspect against high-level requirements
- Verify with the customer where possible
- Repeat steps 2 to 5 until complete and then release

Class diagrams in the requirement-analysis phase contain only application-domain classes. The main stakeholders are end users, customers, and analysts. It is the basis for communication between analysts, application-domain experts, and end users. It is used to bridge the requirement analysis with the design phase.

Class diagrams in the design phase (the object-design model) contain application- and solution-domain classes. The main stakeholders are class specifiers, class implementers, class users, and class extenders. It is the basis for communication between designers and implementers.

Test Yourself

Which of the following usually changes more often than the others?

Entity

Boundary

Control

Test Yourself

In the *ProjectPortal project*, *ListProject* should be a(n) ___ class.

STOP AND CONSIDER

Order the following class relationships from the strongest to weakest.

Number the boxes below to assign each step to its correct place in order.

Steps	Your Order
☐ Association	1.
☐ Composition	2.
☐ Inheritance	3.
☐ Aggregation	4.

Show Answer

Conclusion

In this module, we focused on requirement analysis and discussed how to use user stories to represent and manage requirements. We also discussed the difference between functional and nonfunctional requirements. To help them understand requirements and translate them into software, developers usually use the entity-boundary-control pattern to identify domain objects, and class diagrams to describe detailed requirements.

Bibliography

- Rubin, K. S. (2012). *Essential Scrum: A practical guide to the most popular Agile process*. Addison-Wesley Signature.
- Stellman, A., & Greene, J. (2017). *Head first Agile: A brain-friendly guide to Agile principles, ideas, and real-world practices*. O'Reilly Media.