# SD&D Final Exam

## YACS: An Object-Oriented Analysis

Weston Jones

**This sheet is to certify that Weston Jones in section 1 of Software Design and Documentation did not receive help from any other student on his exam and that his exam is entirely his own work**

Signature of SD&D Student: [illegible] Date: 4/6/19

**This sheet is to certify that Weston Jones in section 1 of Software Design and Documentation visited the Center for Global Communication+Design to review his final exam paper on 4/2/19**

Signature of COMMD Staff: [illegible] Date: 4/2/19

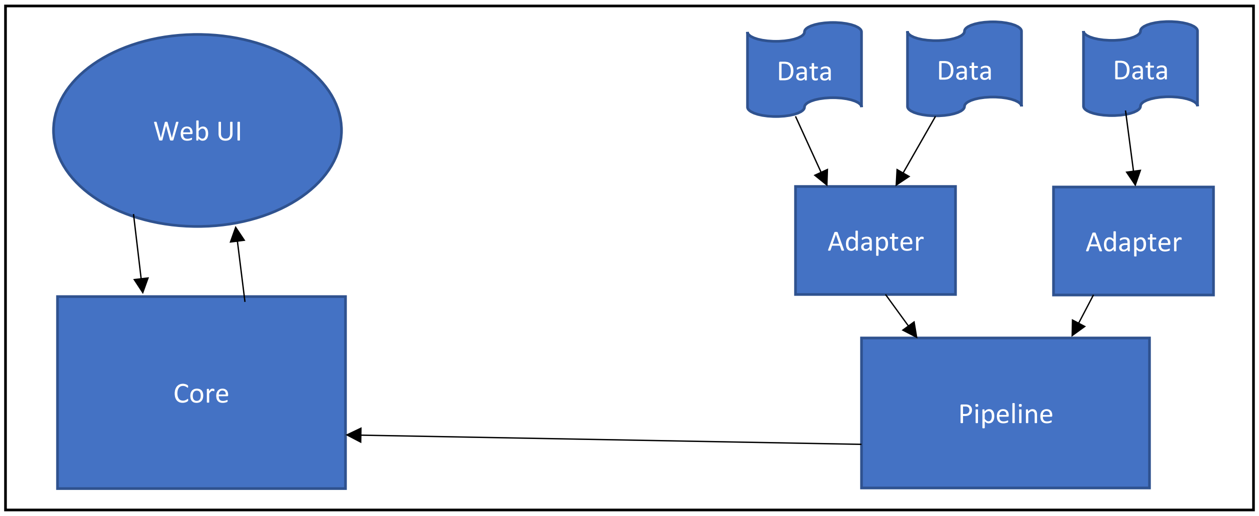
Signature of SD&D Student: [illegible] Date: 4/2/19.

# Question A

## Description

The focus of this paper will be on the open source software YACS (https://yacs.cs.rpi.edu/). YACS is the current iteration in a long line of attempts to make a universal, user friendly, course scheduling app for colleges across the United States. In 2010, the go-to application for this purpose was known as the "Rensselaer Open Course Scheduler." In 2012, this app was scrapped and something new under the name YACS was developed, only to for that, too, to be completely reworked and re-released in the Spring of 2016 (Young, Ada, et al, 2016).

In its current state, YACS is composed of multiple interlocking components, all written in various languages and employing different technologies. First there is what the developers call the "Pipeline." This component functions as a way to read in course data from various sources for later processing. Because YACS is designed to be utilized by any university, the pipeline uses "Adapters" to help it parse data from differing sources. Adapters extract the raw data from course catalogs, pdf files, etc. and feed it to the Pipeline which combines and converts it into a universal, intermediate form – a list of courses and their attributes. This is passed to the YACS "Core" which computes schedules and processes course conflicts for the web interface ("Overview of the Yacs Architecture," 2016).



The web interface is the part of this software that most are likely familiar with. The interface allows users to view the descriptions of courses offered by each department at a university or search for specific courses based off a keyword. Users can select courses to add them to a potential schedule. Once several courses have been selected, a user can see a graphical representation of their selected courses on a weekly schedule. Arrow icons allow users to flip between multiple versions of said schedule (if a course has multiple timeslots, for example).

Many students find this preferable to navigating their university's own course catalogs or registration portals, as these often have very plain, purely text-based interfaces. When designing schedules, it's helpful to see chunks of time displayed on a calendar where one can better consider what their days might be like.

While most scheduling software, such as Google Calendar, Outlook, etc., can all display a graphical view of a user's schedule, YACS is unique in that it comes preloaded with accurate course information. It's a simple of matter of just selecting which courses look interesting, then flipping between possible schedules.

## Feature One

With as solid a foundation as the existing YACS' system, there are many opportunities for adding features. One possibility would be to use data to predict what a future semester's course offerings might look like. By looking at course offerings from past semesters and extending the "Pipeline" to read in information regarding when courses will be offered, YACS could generate possible course selections for any future semester.

The feature could be implemented as a drop-down selection menu in the upper left-hand corner of the main YACS landing page. YACS would load with the current semester selected but allow users to select past semesters or future semesters. From there, after doing some processing behind the scenes to load in the new course offerings, YACS could function as it normally does: allowing users to view course offerings and construct schedules.

The benefits of this feature are twofold. First, students looking to plan ahead would be able to utilize YACS' more pleasant interface to compare possible schedules. This would be easier than using a university course catalog to accomplish the same thing.  For example, RPI's course catalog lacks the intuitive, graphical interface and only tells a user whether a course is generally offered in the Fall or Spring. This feature could help avoid situations where a student has a schedule planned out based off of what courses they know will be offered in the Fall or Spring, but end up having to change things because there ends up being a time conflict between two courses.

Second, this feature would be massively helpful to YACS developers. For RPI at least, course information is read in through the "current" version of course offerings posted a few weeks before class registration starts. This creates a scramble to configure YACS to properly read all this data in and often requires use of administrator functionality to manually correct any errors. With this feature, most of the course offerings would already be generated ahead of time. YACS could then even automatically correct itself when new course offerings are posted, saving time for the developers.

## User Stories

- As a current college student, I want to be able to use the familiar YACS interface to view future semester's course offerings so that I can better plan my degree ahead.

- As a prospective exchange student, I want to be able to view a university's course offerings for a future semester so that I can decide if that university is a good fit for my academic goals.

- As a YACS developer, I want to YACS to automatically populate each subsequent semester's course offerings, so that I can save time getting this data myself.

## User Scenario

Kieran Ferris is a current Computer Science student at the University of Bristol in the UK. Kieran is particularly interested in machine learning, but his university doesn't have any courses on the topic. Kieran thinks he can work around this by studying abroad his next semester at an advanced American tech school.

While reviewing the University of Bristol's exchange partners, Kieran quickly becomes frustrated with the poor organization of the colleges' websites – finding that their course offerings are hidden behind lots of menus and are displayed as a giant list which takes forever to look through.

Eventually Kieran decides to take a look at Rensselaer Polytechnic Institute, a tech school in New York his home university has an exchange partnership with. His home university's study abroad guidance page directs him to a website called YACS to look at possible course options.

Kieran is relieved to find that YACS is exactly what he's looking for: an application that lists departments and the courses they offer. He first clicks a drop-down menu in the upper left-hand corner of the menu (currently set to "Fall 2018" – the current semester) and changes it to "Spring 2019" – the next semester he plans to study abroad during. The page refreshes but the list of departments stays the same. Kieran clicks on "Computer Science" from the list and starts browsing. He quickly finds a course titled "Intro to Machine Learning from Data" which he happily selects. The "Schedule" button in the upper right-hand corner of the page's title bar pulses to reflect the addition. Kieran clicks "Schedule" and is directed to a relatively sparse looking page that has a Monday to Friday calendar with a yellow block in it representing the machine learning class he just selected. Beneath the calendar, he sees the name, description, prerequisites, and other useful information regarding the same machine learning class. Kieran clicks the arrows next to the schedule and observes how the yellow box moves around to reflect the different times the course is offered.

Now with a sense of how the tool works, Kieran navigates back to the course offerings page by clicking the home button in the upper left-hand corner and selects three more courses – Computer Vision (which mentions the use of some machine learning tools in its description), Probability Theory, and Linear Algebra (both of which are mathematical concepts used in machine learning). Kieran once again clicks on the "Schedule" button.

The schedule shows up blank and a message in the center of the screen indicates there's a time conflict. Kieran scrolls down and checks the descriptions of the courses he selected. He sees that Computer Vision and Probability Theory are both offered at the same time. After some thought, Kieran decides that Probability Theory is of more use to him, so he deselects Computer Vision. The schedule refreshes, and Kieran is able to cycle through the various possible schedules with his three courses. Eventually he settles on the one that will require him to wake up the latest and sits back, content.

While the YACS website has warned him that the "future" schedule is purely hypothetical and could change, he checks the course offerings from previous spring semesters via the drop-down menu and sees that all the courses he's chosen have been consistently offered at the same time as they are now and never reached max capacity. Fairly certain that he'll have a curriculum that interests him, Kieran makes the necessary arrangements with his home university to study abroad at RPI next semester.

## Implementation Details

Since YACS is a web application that uses a variety of scripting languages, it doesn't have "classes" in the same way a desktop application written in Java might. Still, for the purpose of this analysis, think of classes as components, services, or modules – chunks of code with a common function.

Moving forward, there are three main parts to this feature. First, there needs to be a way for finalized course offerings from each semester to be stored permanently in a database for later retrieval and processing. Second, the YACS Pipeline and Adapters need to be extended to additionally read in "time offered" information from course catalogs. And third, there needs to be a way to utilize these two new sources of data to predictively generate future course offerings.

The YACS Core should only be responsible for processing schedules and course conflicts for display on the web interface. Rather than tasking this component with more functionality and making it less cohesive, responsibility for these additional functions should be delegated to new components. The component should also be renamed to something more descriptive of its actual function, so it shall now be called the “Scheduler.” It will also be given its own interface to decouple it from the rest of the components and fulfill the Dependency Inversion Principle.

The first new component added is called “Database Operations.” Because this new feature will require multiple components reading and writing to a database, there should be a dedicated component to process those requests. This will create a component with high cohesion and low coupling that will abstract away most of the database connection backend. To create further abstraction and fulfill the Dependency Inversion Principle, this component will also have its own interface.

Second, generating future lists of course offerings will require lots of computation, so there should be a dedicated component to handle this. A new component called the “Processor” will take “time offered” information from the Pipeline component, compare it against past semester course offerings already in the database, through some calculation algorithm generate future semester course offerings, and then store these new course offerings back in the database.

Third, the Pipeline should be given its own interface and extended with the ability handle “time offered” information. It should expect to receive this information from an Adapter and generate course lists that include “time offered” among the attributes describing each course.

Lastly, there should be a universal Adapter interface -- allowing the creation of multiple Adapter components. Each Adapter will read from potentially different data sources potentially different kinds of course attributes but will process it into a standardized format so that it can be used by the Pipeline polymorphically.

## CRC Cards

To simplify the CRC cards, trivial interfaces that exist solely to fulfill the Dependency Inversion Principle will be omitted. Assume that when a component collaborates with another component, it does so through an interface. The exception to this rule is the Adapter Interface, which is non-trivial because it allows adapters to be polymorphically used by the pipeline to construct comprehensive lists of courses. The Processor component also does not have an interface because no component depends on it or uses its functions.

| **Database Operations** | |
|---|---|
| Cohesiveness: 7 (Informational Cohesion) | |
| Responsibilities: | Collaborators: |
| - Process and execute requests to write course information for storage in a database.<br>- Process and execute requests to update record(s) regarding course information in a database.<br>- Process and execute requests to read and return information regarding courses. | - Processor (Coupling: 5)<br>- Web UI (Coupling: 5) |

Cohesiveness: The Database Operations component gets a 7 because all its responsibilities solely involve interacting with a single data structure (the database). While this component’s functions might share some connection overhead for efficiency, each function (read, write, modify) can be delineated.

Coupling: All collaborators scores low for coupling because the Database Operations component simply responds to requests from the Processor and Web UI and returns the requested data or performs the requested operation. The details of the actual database connection are abstracted.

| **Processor** | |
|---|---|
| Cohesiveness: 6 (Functional Cohesiveness) | |
| Responsibilities: | Collaborators: |
| - Receive intermediate data from the pipeline<br>- Look for an existing course ID in the database and update the record with the new data.<br>- If no existing course ID is found, create a new record with the new data received. | - Database Operations (Coupling: 5)<br>- Pipeline (Coupling: 5) |

Cohesiveness: The processor component gets a 6. This component has a bit more to do (both receiving data from the Pipeline and matching it against data already in the database via the Database Operations component), and its functionality spans multiple data structures (the raw data received from the Pipeline in addition to the database). Still all its functionality is related to a common task and can makes sense to be grouped together.

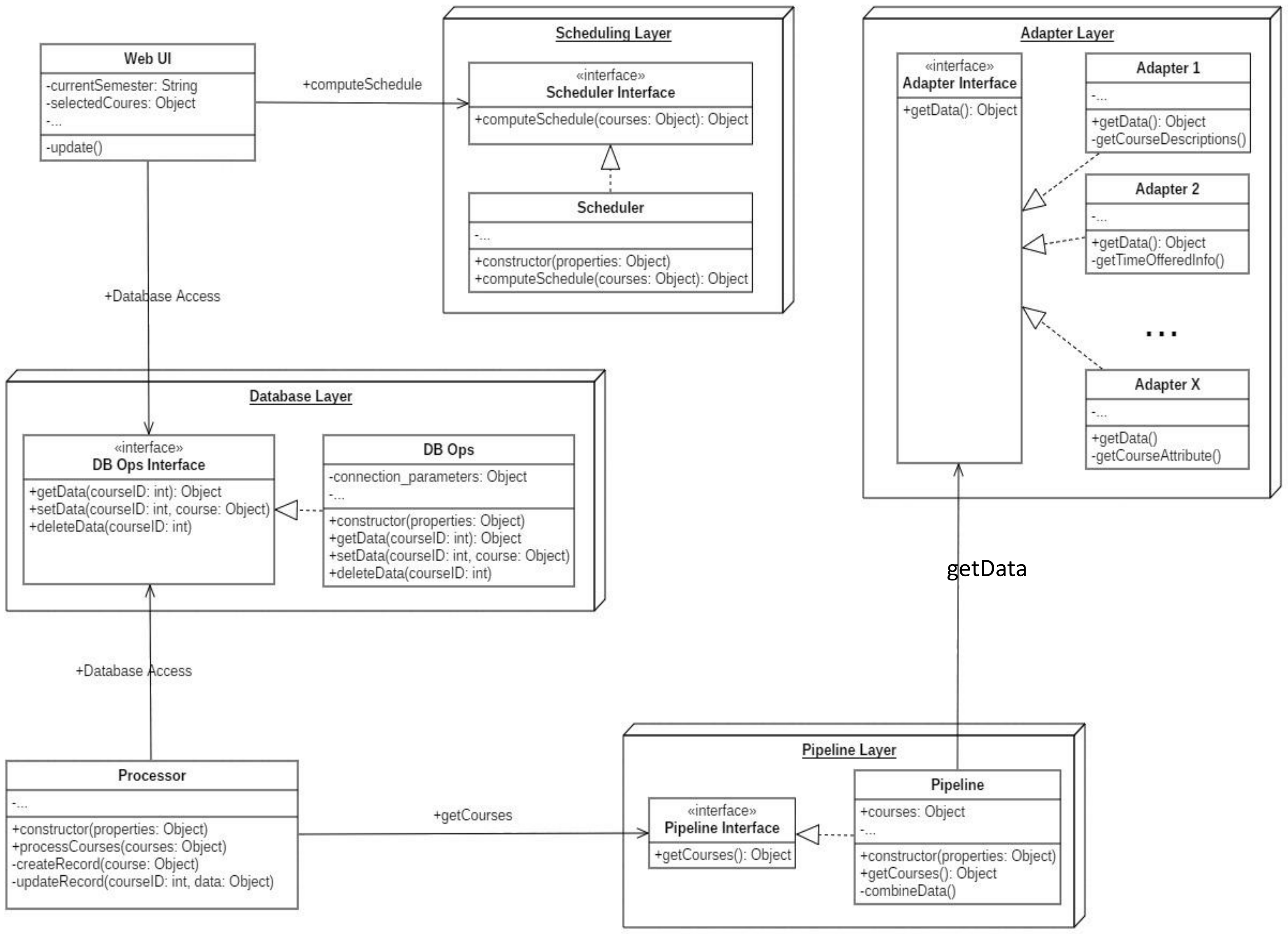
Coupling: All collaboration scores low for coupling because the Processor simply receives raw data from the Pipeline and just calls read() and write() on Database Operations.

| **Adapter Interface** | |
|---|---|
| Cohesiveness: 7 (Informational Cohesiveness) | |
| Responsibilities: | Collaborators: |
| - Provide a universal interface allowing the Pipeline to retrieve course information. | - Pipeline (Coupling: 5) |

Cohesiveness: The Adapter Interface gets a 7. Adapters can potentially read in different types of data, but all of it goes towards building a single data structure that the Pipeline can read form.

Coupling: The adapter's only collaboration involves eventually returning data to the Pipeline that follows a specified format.

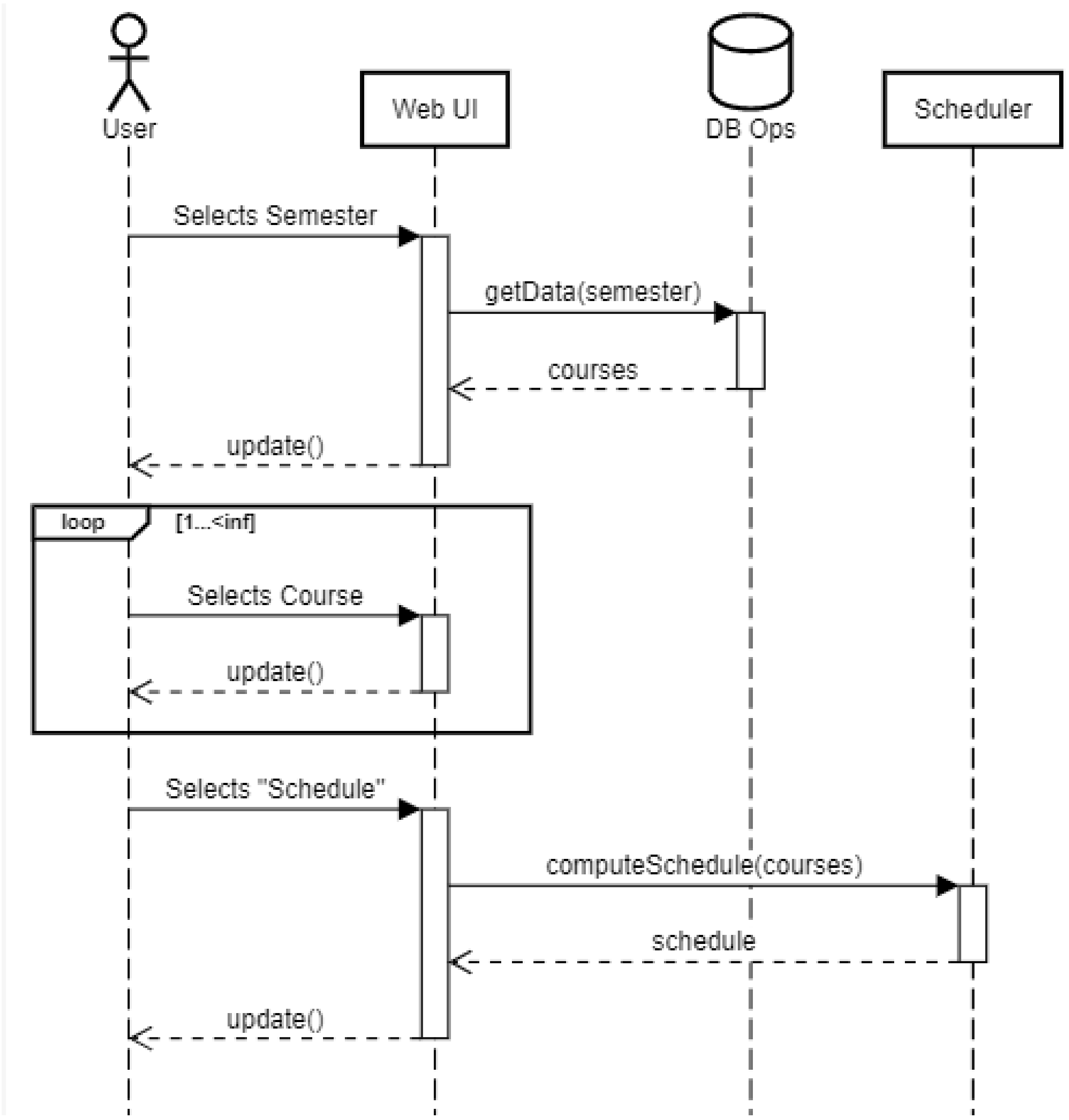
Class Diagram



The diagram above is a visual representation of the implementation details discussed earlier. For clarity, related components and their interfaces are bundled into "Layers." Arrows denote interface realization and directional associations (This component calls this function in another component).

## Sequence Diagram

This sequence diagram represents an interaction a user might have with the “front end” of this feature: Selecting a semester, viewing courses, and making a schedule. Arrows show actions performed by one component on another and the response or data returned.



## Feature Two

College students will often make what are colloquially called “four-year plans” – spreadsheets that show the breakdown of what classes they want to take and when over the course of their four years at an institution. They can then compare this against whatever guidelines their university has put out that list degree requirements to make sure they’re on the right track to graduating (see example).

| **Freshman Year** | | **Sophomore Year** | |
|---|---|---|---|
| **Fall Semester** | **Spring Semester** | **Fall Semester** | **Spring Semester** |
| CSCI 1100 Computer Science I | CSCI 1200 Data Structures | CSCI 2200 Foundations of Computer Sci | CSCI 2300 Introduction to Algorithms |
| PHYS 1100 Physics I | IHSS 1220 It & Society | CSCI 2500 Computer Organization | CSCI 2600 Principles of Software |
| MATH 1010 Calculus I | MATH 1020 Calculus Ii | MATH 2010 Multivariable Calc & Matrix Algebra | ITWS 1100 Intro to IT & Web Science |
| ECON 1200 Introductory Economics | ECON 2010 Intermediate Microeconomic Theory | ECON 4140 Structure of Industry | ECON 2020 Intermediate Macroeconomic Theory |
| **Junior Year** | | **Senior Year** | |
| **Fall Semester** | **Spring Semester** | **Fall Semester** | **Spring Semester** |
| CSCI 4430 Programming Languages | CSCI 4440 Software Design and Documentation | CSCI 4100 Machine Learning from Data | CSCI 4150 Introduction to Artificial Intelligence |
| CSCI 4220 Network Programming | CSCI 4210 Operating Systems | | |
| ITWS 2110 Web Systems Development | ITWS 4500 Web Science Systems Development | ITWS 4310 Managing IT Resources | ITWS 4370 Information System Security |
| MATP 4600 Probability Theory and Applications | BIOL 2210 Introduction to Cell Biology | MATH 4100 Linear Algebra | ARTS 1020 Media Studio: Imaging |

With the existing YACS Scheduler, a way to utilize historical course offerings and predict future ones, and the addition of a system that can read in degree requirements from a university’s catalog, YACS could be extended into a tool that makes this process much easier and more accurate.

Similar to how users select courses, a separate “Four Year Plan” interface could let users select degrees they wish to complete. From there, they would be redirected to a page with a grid of schedules corresponding to different semesters and with dedicated buttons for adding, removing, and modifying the courses taken or to be taken in said semesters. Upon selecting courses for a semester, YACS would “check off” requirements of the user’s chosen degrees.

Four-year plans are valuable in that they are useful tools for planning ahead and are sometimes required for graduate or co-term students. A tool that allows for their easy generation – both with an interface that is easy to navigate and has necessary information already built it – would be invaluable to students.

## User Stories

- As a current college freshman with a declared major, I want to be able to quickly generate a four-year plan with accurate information so that I can better plan my degree ahead.

- As a current college junior, I want to see how adding a second degree would affect my future semester course load so that I can better plan ahead.

- As an academic advisor, I want to create and view sample four-year plans, so that I have more reference material to help my students.

## User Scenario

Malcolm Harper is a current second semester freshman at RPI studying Computer Science. While Malcolm likes Computer Science and wants to pursue it, he also has an interest in Biology and Bioinformatics. Because he started college with a semester's worth of AP credit, and because of the slight overlap with Bioinformatics and Computer science, Malcolm wants to declare a second major and still graduate in four years.

Malcolm remembers using the YACS application when he was first choosing classes and recalls that it now has a built-in function to construct four-year plans, which he thinks will be helpful now. He goes to the YACS website and clicks on the "Four Year Plan" button in the upper right-hand corner, right next to the "Schedule" button. He is brought to a page similar to the YACS homepage with all the departments at RPI listed, however this time, rather than displaying courses when he selects a department, he sees a list of degrees offered. He selects the Bachelor of Computer Science and Bachelor of Bioinformatics degrees under their respective departments.

The new page he's brought to has a 4 x 2 grid of empty blocks representing a typical four-year, eight semester college timeline. Underneath, the requirements for the two degree are displayed. He clicks on the first block corresponding to Fall 2016 – his first semester at RPI – which navigates him to a view of YACS that he's used to seeing. Departments at RPI and their offered courses are displayed, but there are now dedicated "Cancel", "Home", and "Confirm" buttons centered in the title bar of the page in addition to the usual "Schedule" button in the right-hand corner.

Malcolm selects the courses he took his first semester then clicks the "Confirm" button. He is navigated back to the 4 x 2 grid page, only now the first block is filled in with all the courses he just selected. He repeats the process for the second block (his current semester).

The courses he's already taken are checked off from the list of degree requirements below. He also takes this time to manually check off a few general requirements himself to account for the AP credits he's taken. As Malcolm goes to fill out future semesters, the application hangs a little bit as it processes future schedules and sends a message that lets him know all of this is hypothetical. Still, Malcolm finds the entire experience quite helpful – he learns that Chemistry II and Introduction to Algorithms are generally offered on Monday and Thursday mornings at the same time, and so he adjusts his schedule accordingly – electing to take Algorithms a semester earlier.

When he is finally done, Malcolm sees a four-year plan that checks off all of his degree requirements that he's reasonably confident he could follow. He hits the export button to generate a neat pdf with the same information which he then sends in an email to his advisor for feedback.

## Implementation Details

Degree requirements are a fundamentally different datatype than courses. Trying to integrate functionality to handle this new data format into the existing Adapter to Pipeline to Processor track would lower the cohesion of all those components, so creating new components designed to deal with this data is a better option.

First, there should be new tables for storing offered degrees and their requirements within the database, but the existing Database Operations class can be used for reading and writing to them. It lowers the cohesion of the component a little bit, but keeping the connection overhead confined to a single component is worth the tradeoff.

Next, the Adapter Interface created in the first feature can be reused. Adapter components that implement the interface will handle the new “requirements” datatype. Additional Adapters will have to be created to read data that can be used to assign each class an attribute that can be used to determine whether a class fulfills a certain requirement (Using subject codes would probably be the best way to accomplish this).

For the Pipeline, there should be a separate component called the Planning Pipeline (with a corresponding interface). To avoid confusion, the Pipeline and Pipeline Interface discussed in the first feature will now be referred to as the Course Pipeline and Course Pipeline Interface. The Planning Pipeline will function similarly as before, just processing degree requirements.

The Processor will get similar treatment. A new component called the Planning Processor used for updating the database when new degrees and/or requirements are received from the Planning Pipeline will be created with its own interface, and the old Processor renamed to the “Course Processor” to better differentiate the two.

Finally, there needs to be a new component called the Planner (also with a corresponding interface). This component will handle the logic for “checking off” currently selected degree requirements given a list of courses currently in a four-year plan.

## CRC Cards

As before, trivial interfaces that simply serve as abstractions to decouple code and fulfill the Dependency Inversion Principle will be omitted from the CRC cards for the sake of simplicity. Assume that collaboration between components occurs via interfaces.

| **Planning Pipeline** | |
|---|---|
| Cohesiveness: 6 (Functionally Cohesive) | |
| Responsibilities: | Collaborators: |
| - Read and combine information from adapters into a cohesive intermediate data structure.<br>- Send finished data structure to the planning processor. | - Adapter Interface (Coupling: 5)<br>- Planning Processor (Coupling: 5) |

Cohesiveness: The Planning Pipeline takes in data from multiple sources and pieces it together. The functions are related but not entirely cohesive.

Coupling: The planning pipeline’s only interactions with other components just involves the transfer of data.

| **Planning Processor** | |
|---|---|
| Cohesiveness: 6 (Functionally Cohesive) | |
| Responsibilities: | Collaborators: |
| - Read data from planning pipeline.<br>- Create new entries in database for degree requirements.<br>- Update existing entries in database for degree requirements. | - Planning Pipeline (Coupling: 5)<br>- Database Operations (Coupling: 5) |

Cohesiveness: The planning processor gets a 6 (functional cohesiveness). Like the course processor discussed in feature one, this component works with multiple datasets – both the raw data from the pipeline and the database through the operations class. Still its functionality is all related and works toward the common goal of correctly placing data from the pipeline into the database.

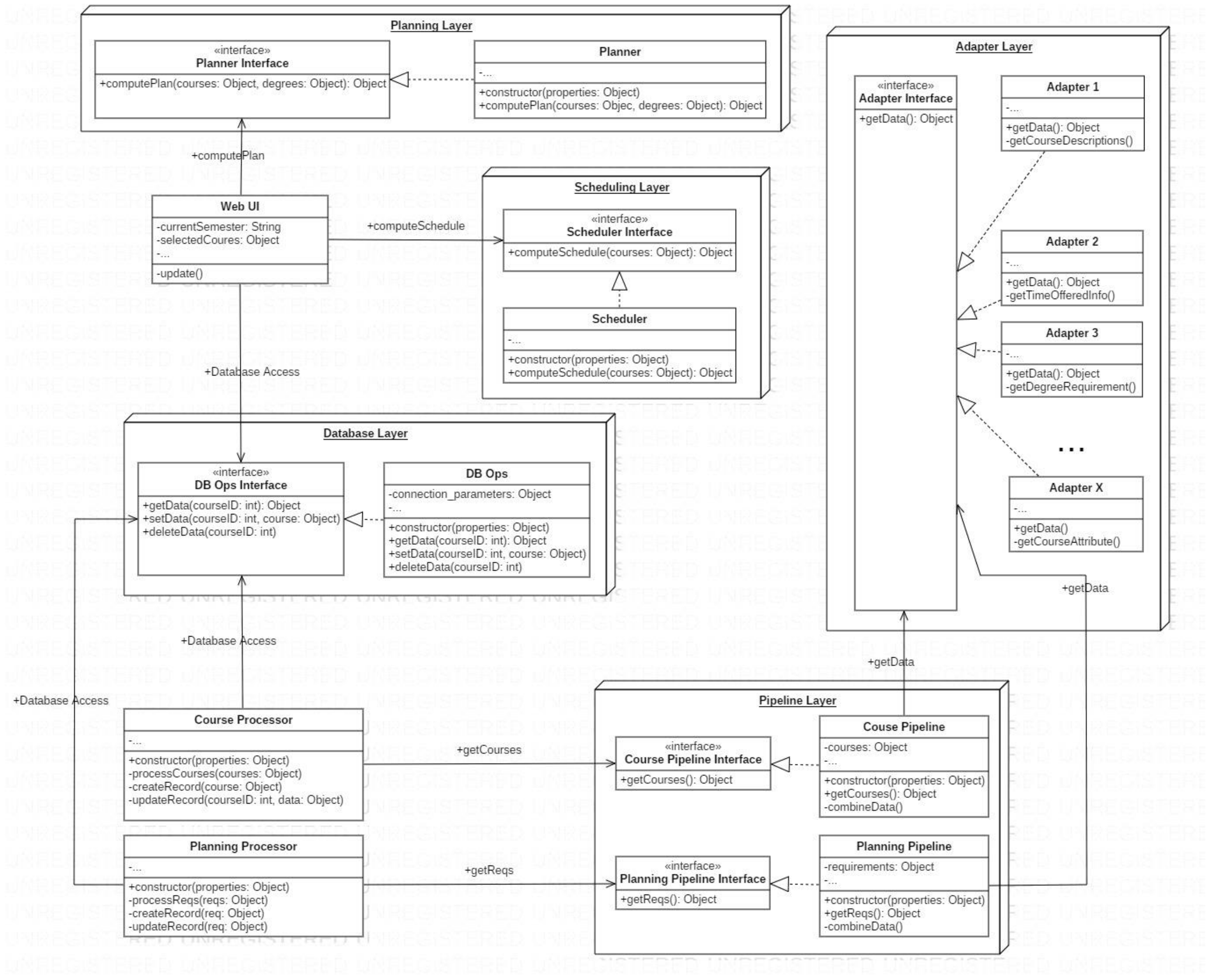
Coupling: The planning processor scores 5 for coupling because it's only collaboration with other components just involves sending and receiving data.

| **Planner** | |
|---|---|
| Cohesiveness: 7 (Functionally Cohesive) | |
| Responsibilities: | Collaborators: |
| - Check off requirements given a list of selected courses | - Web UI (Coupling: 5) |

Cohesiveness: The Planner takes in multiple datasets (selected degree requirements and selected courses) but accomplishes the single function of applying those courses to the requirements.

Coupling: The Web UI will call upon the Planner to check off requirements given degree requirements and selected courses. The Planner simply returns which requirements have been checked off.

## Class Diagram

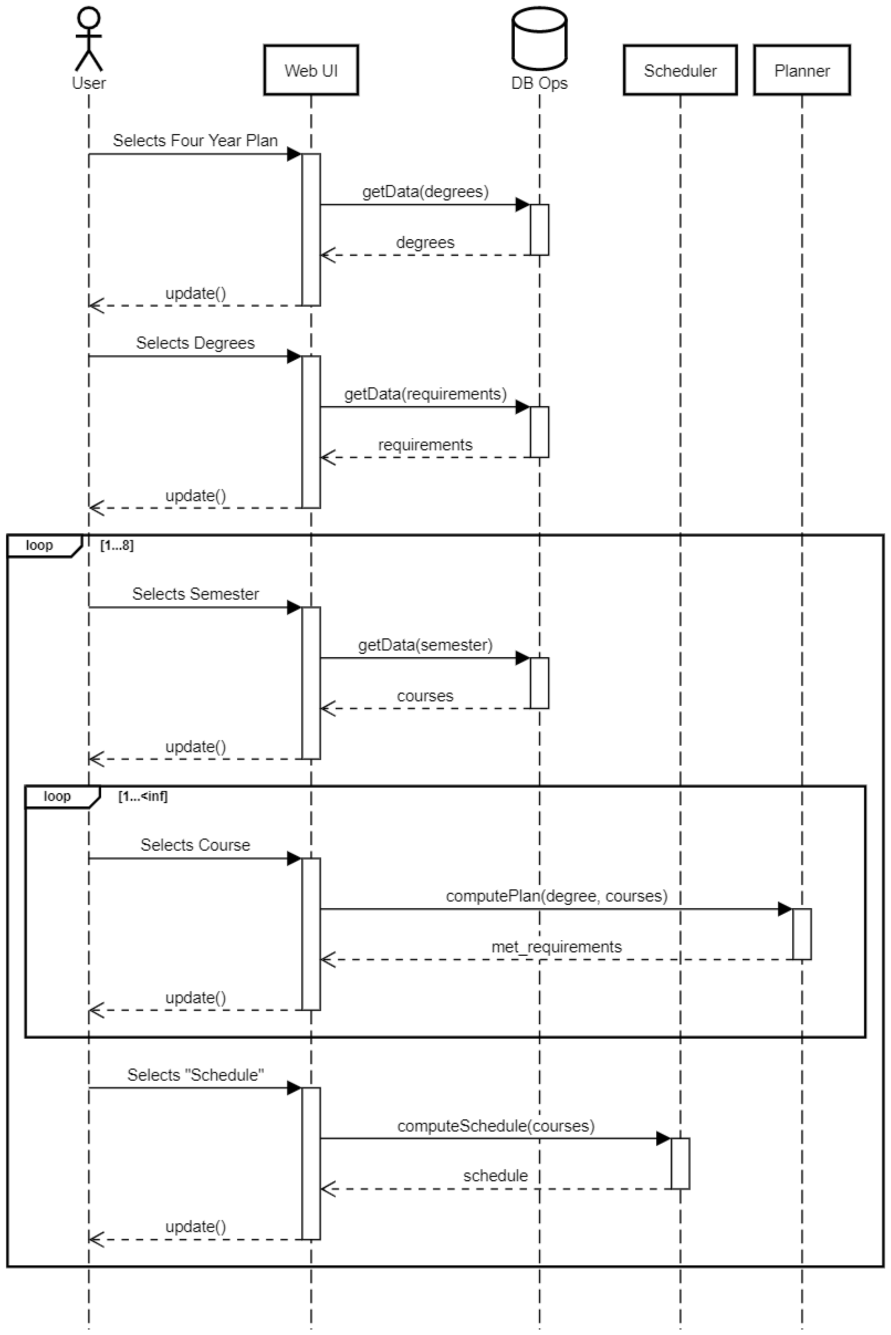


Planning Layer
«interface»
Planner Interface
+computePlan(courses: Object, degrees: Object): Object
Planner
-...
+constructor(properties: Object)
+computePlan(courses: Objec, degrees: Object): Object
+computePlan
Web UI
-currentSemester: String
-selectedCoures: Object
-...
-update()
+computeSchedule
Scheduling Layer
«interface»
Scheduler Interface
+computeSchedule(courses: Object): Object
Scheduler
-...
+constructor(properties: Object)
+computeSchedule(courses: Object): Object
+Database Access
Database Layer
«interface»
DB Ops Interface
+getData(courseID: int): Object
+setData(courseID: int, course: Object)
+deleteData(courseID: int)
DB Ops
-connection_parameters: Object
-...
+constructor(properties: Object)
+getData(courseID: int): Object
+setData(courseID: int, course: Object)
+deleteData(courseID: int)
+Database Access
+Database Access
Course Processor
-...
+constructor(properties: Object)
-processCourses(courses: Object)
-createRecord(course: Object)
-updateRecord(courseID: int, data: Object)
Planning Processor
-...
+constructor(properties: Object)
-processReqs(reqs: Object)
-createRecord(req: Object)
-updateRecord(req: Object)
+getCourses
+getReqs
Pipeline Layer
«interface»
Course Pipeline Interface
+getCourses(): Object
Couse Pipeline
-courses: Object
-...
+constructor(properties: Object)
+getCourses(): Object
-combineData()
«interface»
Planning Pipeline Interface
+getReqs(): Object
Planning Pipeline
-requirements: Object
-...
+constructor(properties: Object)
+getReqs(): Object
-combineData()
Adapter Layer
«interface»
Adapter Interface
+getData(): Object
Adapter 1
-...
+getData(): Object
-getCourseDescriptions()
Adapter 2
-...
+getData(): Object
-getTimeOfferedInfo()
Adapter 3
-...
+getData(): Object
-getDegreeRequirement()
...
Adapter X
-...
+getData()
-getCourseAttribute()
+getData
+getData

The diagram above is a visual representation of the implementation details discussed earlier. For clarity, related components and their interfaces are bundled into “Layers.” Arrows denote interface realization and directional associations (This component calls this function in another component).

## Sequence Diagram

This sequence diagram shows how a user will interact with the “front end” of the second proposed feature. A user selects the “four-year plan” option then selects courses for each of the eight semesters they will have at an institution. Arrows show actions performed by one component on another and the response or data returned.



## Third Feature

A sign of good design is if one can add additional features to a project that give lots of "bang for the buck" – they add lots of value and function while not requiring extensive reworking of existing classes. In the context of this new extension of YACS, any new attribute added to describe a course, such as a teacher review, difficulty ranking, or required textbooks, can be seamlessly integrated into the application.

Consider the course pipeline, which is designed to be agnostic to whatever sort of data it needs to parse, relying on adapters that implement the adapter interface to convert any data to a standardized form. It would be a simple matter to make another adapter instance that could read in additional attribute information about a course – either from another section of the course catalog or perhaps an entirely different website such as RateMyProfessors.

Regardless, the course pipeline can integrate that new attribute into the database records. From there, every other part of the code just treats courses as objects with various properties. When listing course descriptions on the homepage for example, these new properties can be dynamically pulled from the database and displayed.

## SOLID Principles:

The SOLID Principles are a set of formal rules that when followed, generally lead to the development of well-designed and maintainable code. The S in SOLID stands for the Single Responsibility Principle which states that every class or component in software should have a single core function – it's essentially a measure of how cohesive and uncoupled each part of the code is. Small classes that have a single function or a few related functions generally adhere to this principle.

The extension of YACS discussed above certainly adheres to the Single Responsibility Principle. Every component in the design is at the very least functionally cohesive and components are uncoupled from one another. Take for example the separate components for dealing with courses and degree requirements. Delegating a separate chain of components to deal with the two different types of data ensures each component maintains a single, cohesive function.

The O in SOLID stands for the Open-Closed Principle. This is essentially a measure of how well components in a piece of software are abstracted and parametrized. Components that accept a wide range of parameters to carry out a related function on lots of potential inputs generally satisfy this principle.

The design discussed in this paper adheres to the Open-Closed Principle. The functions of all the components have signatures that accept different parameters to customize their functionality while abstracting their implementation. Take for example the functions with the Database Operations component. Each function reads from, writes to, or modifies a particular record specified by the calling component.

The L in SOLID stands for the Liskov Substitution Principal. This principal simply states that "Subtypes must be substitutable for their base types." The subtype must always have a stronger specification than the base class – requiring less and providing more ("Liskov Substitution Principle," 2019). Because of the frequent use of interfaces in the design, subclasses are forced to implement all the methods that a collaborating class might depend on, thereby enforcing adhere to the principle.

The I in SOLID stands for the Interface Segregation Principle which states that if a component depends on another one, the dependent component should use all the functionality of the base component. If it does not, then it is essentially relying on functionality it does not even use, which is bad design and adding coupling where none should exist. This is not a problem with the above design. Every component has a singular functionality which, if collaborating with another class, is used entirely by the collaborating class. One might argue that the Database Operations component, which accesses two different tables within the database when a component might only need access to one, violates this principle, but the function for accessing the table is the same in both cases and centralizing the database connection overhead to a single component is worth the tradeoff.

Lastly, the D in SOLID stands for the Dependency Inversion Principal. This principle states that components should only depend on abstractions, thus forcing components to rely on interfaces (which generally don't change) rather than other components (which change all the time). This principle was tricky to implement, but by adding interfaces to all the main components in the design, every component is essentially decoupled from one another.

## Ease of Change Test

To add these features to the existing YACS application, at least six unique components have to be created (Planner, Database Operations, Course Processor, Planning Processor, Planning Pipeline, and at least one Adapter). This is not counting the extra interfaces those components may use or the fact that there may be an arbitrary number of Adapter components.

To add these features to the existing YACS application, three existing components needed to be changed (The Core/Scheduler, the Web UI, and the Pipeline / Course Pipeline).

## Question B

### Methodology Overview

In response to the need for software development methodologies better equipped to deal with changes in customer requirements, the Agile approach was developed. Agile software development emphasizes iterative development – delivering value to customers in chunks – thereby allowing teams to evolve with their customer's changing wants and needs. Under the umbrella of the "Agile Approach" and of particular interest is the Scrum methodology, which could be good strategy for implementing the features discussed in Question A.

Scrum is an agile development methodology that focuses on short, fixed-length "sprints" of coding that deliver some sort of value to a client, open communication between all team members that is facilitated by distinct user roles, and the presence of "backlog" artifacts (lists of tasks to be completed at various stages in the project's development).

When first assigned a project and before any actual programming begins, a development team following Scrum will meet to establish a product backlog – a prioritized list of deliverable features to be included in the project. To derive this backlog, the team will start with a list of requirements from a customer. From that, they break the requirements into epics – collections of user stories that all relate to a common functionality (Drumond, 2019). These epics are further broken down into individual stories and features. The product manager – a member of the development team who serves as the liaison between the developers and the customers – will listen to feedback from both sides to sort this list based on priority, at which point it becomes the backlog.

With the product backlog established, the development team begins "sprinting." Each sprint also has a planning stage associated with it. Team members will select items (generally those with highest priority) from the product backlog and add it to the sprint backlog. If the product backlog is the overarching "to-do" list, the sprint backlog is the "working list" – a list of tasks to accomplish in the current sprint that, when completed, can be bundled into a shippable feature of the product that the customer can use. To help with this planning is a developer designated as the Scrum Master. This is a person familiar with the unique process of Scrum who is responsible for organizing many of the special meetings integral to the Scrum process.

After establishing the current sprint backlog, the team will discuss how they will complete each item over the course of the sprint (generally a time period of two to four weeks), before finally starting to code. At the start of every day during the development stage of a sprint, a special meeting called the daily standup or daily scrum is held. Each member of the team quickly presents what they've done so far, what they're working on, and if there are any problems or risks they've encountered. If there are big problems, the sprint goals can be changed retroactively, but this is generally not best practice (Drumond, 2019). Daily scrums are when most of team communication takes place, so team members are expected to be transparent and honest with their status reports.

At the end of a sprint there are two additional important meetings that take place. First is the Sprint Review. The goal of this meeting is to congratulate everyone on the progress they've made, formally take stock of where the development of the project is at, and decide whether or not the items completed from the sprint backlog should be released to the customers. The next end-of-sprint meeting is known as the Sprint Retrospective. This is a more internal, introspective meeting that lets the team discuss what could be done differently during the next sprint. Time allocation, communication strategies, and development tips are all examples of possible discussion topics.

After the retrospective, the sprint is formally complete. The team moves on to the next sprint – planning which features should be added to the sprint backlog, planning how those features will be implemented, developing while holding daily scrums, to finally having the end of sprint meetings. This cycle is repeated until all the requirements the customer has are fulfilled and the project is complete.

## Project Plan

| No. | Length | Sprint Goals | Deliverables |
|---|---|---|---|
| 1 | 2 Weeks | - Setup Database to store course information.<br>- Create Database Operations Component.<br>- Create Database Operations Interface.<br>- Manually populate courses table with multiple semesters of course data.<br>- Testing of the Database Operations component | - Working database that is and can be further populated with courses and their attributes. |
| 2 | 2 Weeks | - Add Select Semester Button to existing UI.<br>- Add functionality to the select and display courses offered in a corresponding semester from the database.<br>- Adapt “Core” into “Scheduler” and verify that it works.<br>- Testing of the Web UI and Scheduler. | - Web Interface that displays and schedules courses from multiple semesters.<br>- Database that can store courses but must be manually updated. |
| 3 | 2 Weeks | - Identify a university course catalog that would be good for testing adapter to pipeline functionality.<br>- Write the several adapters to read descriptions, prerequisites, time offered, and other useful data from the chosen course catalog.<br>- Testing of all written adapters. | - Adapters that can extract raw course data from a university course catalog page. |
| 4 | 2 Weeks | - Decide on a universal standardized format to organize course data and attributes.<br>- Write the Course Pipeline component to utilize the data from the adapters.<br>- Testing of the Course Pipeline | - A course pipeline that can amalgamate course data from potentially different sources into a useful, cohesive structure. |
| 5 | 2 Weeks | - Empty the courses table in the database<br>- Write the Course Processor to match information received from the pipeline against information already in the database or create new records.<br>- Testing of the database functionality in the Course Processor | - A functioning pipeline to processor stream that can populate a database with course information from a university catalog. |
| 6 | 2 Weeks | - Experiment with different algorithms that weigh attributes differently to get a reasonably accurate process for predicting future course offerings. | - A processor that can predict when courses should be offered in future semesters. |

| | | | |
|---|---|---|---|
| | | - Write functionality in the Course Processor to update the database to reflect these predictions.<br>- Testing of the complete course processor. | |
| 7 | 2 Weeks | - Identify a university course catalog that has degree requirements listed.<br>- Write an adapter to parse these requirements.<br>- Write or extend an adapter to categorize courses so that they be said to fulfill those requirements. | - Adapters that can strip the text off a university degree requirements page.<br>- Adapters that can use subject codes, course descriptions, or other properties to add category attributes to courses. |
| 8 | 2 Weeks | - Decide on a universal, standardized data structure to organize degree requirements information.<br>- Write the planning pipeline that combines data from adapters looking at degree requirements.<br>- Integrate the "requirement fulfilling" attributes for courses into the Course Pipeline.<br>- Testing of new adapters.<br>- Testing of Planning Pipeline | - A functioning planning pipeline that produces cohesive data structures composed of degrees and their requirements. |
| 9 | 2 Weeks | - Code the planning processor, which stores degrees and their requirements into the database.<br>- Testing of the planning processor. | - A complete adapter to pipeline to database track for degrees and their requirements. |
| 10 | 2 Weeks | - Code the Planner which returns lists of checked off requirements given degrees and selected course.<br>- Final integration and testing. | - The finished design with the features described above. |

## Question C

In Chapter 2 of *The Mythical Man Month*, Fredrick P. Brooks identifies what he calls the "Tar Pit Problem" – the tendency for large software development projects to be delayed, run overbudget, not function as intended, or fail in some other key way. Just how a prehistoric beast caught in a tar pit would further entrench itself through its escape efforts, software development teams experiencing difficulties often employ solutions that don't work or even make the problem worse.

To further analyze the "Tar Pit Problem," Brooks discusses many of the chronic issues he sees with modern software development. One can synthesize these issues into two main shortcomings: the failure to properly plan a project and the failure to account for overheard. Thus, to better gauge the possibly of a project's success, one can ask the development team five questions that touch on these two main issues.

1. What feature or deliverable of the project do you think will be the most difficult to complete? How have you planned around this so that it gets finished on time?

The value of asking this question relates back to the issue of planning. In the section titled "Gutless Estimating," Brooks argues that "It is very difficult to make a vigorous, plausible, and job-risking defense of an estimate that is derived by no quantitative method, supported by little data, and certified chiefly by the hunches of the managers" (Brooks, 2013). When asked how much time a task will complete, software developers often estimate in a perfect world where there are no major roadblocks and progress towards a goal continues smoothly. This, and the desire to appear like a competent programmer capable for quickly completing tasks, drives developers to chronically underestimate the amount of time a certain task will take. This leads to setbacks and improper allocation of time.

By asking this question early on and particularly about an important feature of the project, developers are forced to really consider what a task will entail, how they're going to work through every aspect of its implementation, and how they should plan development around said task. Since there is no purely scientific way to derive an accurate estimate of "time until competition," the best that can be done is encourage critical thinking to deliver better estimates, thus allowing proper timelines to be constructed.

Well thought out answers to this question would show that the development team has broken the project down into tasks, considered the priority and importance of each component, and planned appropriately, thereby allowing them to create a timeline that will better reflect how things work out in reality.

Overly optimistic, poorly thought out, or less concrete answers to this question are indicative of poor planning. To correct this, one would want the development team to do more planning and consider the weight, importance, and priority of each component in the project. Breaking down the project into user stories then playing several rounds of planning poker to foster discussion would be a good technique to use.

2. How will you deal with setbacks in project goals?

Setbacks are almost inevitable with every long-term software development project, but they don't have to be a huge problem if there are mechanisms in place and pre-allocated time to deal with them. Absent a good response however, setbacks in a project's development can snowball into a catastrophe. In the section of *The Mythical Man Month* titled "The Regenerative Scheduling Disaster," Brooks observes that often times, when confronted with setbacks in developments, managers will simply throw more manpower at the problem. This solution fails to account for the overhead of taking the time to let the extra workers become well acquainted with the problem and repartitioning work amongst them. Brooks argues that this will cause additional delays that will cause the issue to be resolved no sooner than if it were handled solely by the original team (Brooks, 2013). The project arrives just as late, but the team will have the tradeoffs of reassigning extra workers -- a disastrous outcome for sure.

Answers to this question that acknowledge the "Regenerative Scheduling" phenomenon, mention time built into the schedule to account for setbacks, and discuss response mechanisms that emphasize transparent and honest communication about what the issue is and what needs to happen are all good.

Answers that may consider employing extra workers in response to setbacks, dismissiveness about the presence of setbacks altogether, or the suggestion that deadlines be pushed back until the issue is resolved, show a lack of thought and are a setup for disaster.

To resolve this, one would want to explain that setbacks will probably happen at some point in the project's lifecycle and that open, clear communication is key to help the team plan an effective response.

3. How will you manage inter-team and intra-team communication?

Throughout the chapter, Brooks talks about communication as being a major source of overhead. Consider the time spent communicating an idea to another person and verifying that they understand your meaning. Now consider the time spent communicating the same idea to two, four, or perhaps eight people. Since everyone interprets information at different rates, processes things at different speeds, may not be totally paying attention, or may ask questions that can raise other questions, communication in large groups can be difficult. And since everyone in a team has to communicate with everyone else in the team, larger teams spend exponentially more time communicating than smaller ones do. Still, large teams are necessary to complete large projects, so effectively managing communication becomes an important issue.

Teams that use a single, standardized method of communicating, like a Slack channel with dedicated subgroups or a Kanban board will generally spend less time getting everyone up to speed. Additionally, having a standard time and place for meetings and check-ins helps things flow naturally.

If teams have multiple means of communication between multiple arbitrarily constructed groups of people, there will be those left out of the loop and more time will be spent getting them caught up.

To help with team communication, the project overseer can establish communication channels (such as a Slack channel) ahead of time and make it company policy that all important exchanges take place there.

4. How will you test key features of the product? Why in that way?

Many software developers generally don't test code as early or as extensively as they should. Preemptively catching bugs and resolving them is much easier than sifting through a mostly complete project to try and pinpoint the source of a specific error. In large projects that haven't been adequately tested, the latter sort of scenario can often cause massive delays and lead to the release of bugged code.

Asking about testing early on cements the idea that testing is a necessary and vitally important part of the development effort and jumpstarts the process of generating tests to ensure key features work as intended.

Answers to this question that identify the most important parts of code and mention specific ways to test those parts all show commitment to thoroughly testing a project.

If the answer to this question suggests that the team won't start testing until the project has been completed, then action needs to be taken to ensure all team members test their code as it is written. To mandate that all members of a development team adhere to a standardized testing policy, code repositories can be figured so that all code must pass a test suite before it can be added to a master codebase.

5. Why do you want this project to succeed. What will you do to ensure that it does?

While not everyone has the spark of intuition that helps them develop the next breakthrough algorithm that solves a long-pondered puzzle in an efficient and novel way, most of what makes a good programmer consists of effort, motivation, and following established best practices. A team composed of unmotivated programmers always expecting someone else to direct them or solve the tough problems will rarely succeed. Asking this question helps nail down exactly what drives a team forward.

If someone is working on a project because they feel personally invested in it or want to see it through, so they have something to be proud of, one can count on them giving lots of effort towards project competition. If someone is working on a project because they've been asked to or because they simply want to get paid, they might not want to go the extra mile sometimes necessary to make sure things succeed. Reminding them of their importance in the grand scheme of the team and that the success of the project reflects on them as well can help motivate them and give more to the team.

## Sources

1. *The Mythical Man-Month Essays on Software Engineering*, by Frederick Phillips Brooks, Addison-Wesley, 2013, pp. 1–21, black.goucher.edu/~kelliher/f2010/cs245/theMythicalManMonth.pdf.

2. Drumond, Claire. “Scrum - What It Is, How It Works, and Why It's Awesome.” *Atlassian Agile Coach*, Atlassian, 2019, www.atlassian.com/agile/scrum.

3. “Liskov Substitution Principle.” *DevIQ*, DevIQ, 2019, deviq.com/liskov-substitution-principle/.

4. “Overview of the Yacs Architecture.” *Yacs.io*, Rensselaer Center for Open Source, 2016, yacs.io/#/architecture/overview.

5. Young, Ada, et al. “About YACS.” *YACS 2019*, Rensselaer Center for Open Source, Mar. 2016, yacs.cs.rpi.edu/#/about.