Reading: EC-06 Enterprise Architecture

EC-08 Digital Engineering

To avoid plagiarism, please provide appropriate references.

Instructions:

1. (12 points) Based on UML and SysML Diagram taxonomies described in EP-08, identify minimum essential diagrams that are necessary for the following project types and provide supporting rationale

*****I used the references [1] and [2] for Q1.*****

A. A small-scale software-intensive project such as a content management system for CS department website or a photo editor app

Since this is a software-intensive project we should use UML diagrams here. I would suggest starting out by using some structure diagrams to illustrate the overall system.

Class Diagram; This Diagram would be essential to fully see all the objects that will have to be programmed, along with their values and how they relate to each other. This will be important so that we build the right system to accommodate the vision of this small-scale project

Object Diagram; This can be very good complimentary diagram for the class diagram, especially for small scale project. Since it is small you should be able o make use of showing some instances of objects to show more context of the class diagram. This would be hard to show for larger projects that may have a lot more class diagrams and more complex structures.

Use Case Diagram; This will be essential to fully illustrate what users of the system can and cannot do. This will help flush out requirements and with small-scale project this could be pretty detailed without creating a useless diagram with too many use cases or too abstract use cases.

Sequence Diagram; This is a good compliment to the Use Case diagram. This diagram will be essential because it will give a clear picture of how the functionality in this system will run.

B. A small to medium scale software-intensive project such as Dropbox, Intuit QuickBooks, Slack

This is a software-intensive project so we should use more UML models here again. I would suggest the following diagrams.

Class Diagram; Just like before this will be an essential to develop a full view of all the object that will be programmed in the project. I think the class diagram will be an essential diagram for most projects regardless of the size because it gives great detail and usefulness for when it comes down to programming the project.

Use Case Diagram; A use case diagram can still be very useful for a medium size project. This will give a good view of requirements and functionality. Since the project is not too big yet the use case can still be useful here.

Component Diagram; I think a diagram like this because it will illustrate the structure of the software system as whole. As you get to a little bit larger project you will benefit more from more abstracted

diagrams like this one. You will already have a Class diagram showing the class relationships with details but you also need to know about decencies and exactly how system components are interacting.

Sequence Diagram; A sequence diagram will still be essential here. A project at this level is still not doing complex transactions so a sequence diagram can still be used here showing a lot of detail without being too much.

C. A large-scale software-intensive project such as Facebook, online banking, office 365

Class Diagram; This should still be used even at a large-scale project. Again this diagram will help in the implementation process by ultimately giving a blue print to what needs to be programmed.

Use Case Diagram; Along side the Class Diagram we should also still use a use case diagram. At this level of a project the Use Case diagram will be more abstract but it will still help in showing some main requirements and functionalities that must be programmed.

Model Diagram and Component Diagram; These two diagrams will be very beneficial for a large-scale project to show the "Big Picture" of the system as a whole. Since the project is so large it can difficult to grasp the complexity of the project and all its moving parts. Thus a Model diagram to show the different sections of the project can be a big help. Additionally the component diagram will help show a more detailed illustration of interaction while still having the abstraction of the project.

Deployment Diagram; A project this large will have some complexity when it comes to deployment. A larger project usually mean a larger user base which means more deployment which adds more risk. Thus a deployment diagram will be beneficial here to clearly illustrate the process of deployment.

D. A small-scale software embedded project such as no contact infrared thermometer, robot vacuum

Since this is a software embedded project we should be using more SysML diagrams here. I would suggest the following diagrams to be used for a small-scale project.

Requirement Diagram; This will be crucial for all scaled projects as it will outline the requirements needed and the relationships between different requirements. No matter what the project is, something like a robot vacuum, you will need to outline requirements and find dependencies.

Block Definition Diagram; This will be useful at defining the structure of the project. This is crucial for showing how other Control, Data, and Interface Objects are used with the system. Similar to the requirements diagram this will be essential to all scale projects.

Activity Diagram; This type of diagram will be essential to showing the process of the system. I think this would be an essential diagram for a small scale project like this to make sure we illustrate the behavior or the diagram. Given that the project is small I think this diagram can nicely capture the entirety go the behavior.

E. A small to medium scale software embedded project such as smart electric vehicle charging station, Amazon Alexa

A medium sized project will benefit a lot from similar diagrams as the small scaled project. However, they can be more detailed and more robust.

Requirement Diagram; This will still be very essential for a project this size. Showing the requirements and their relationship to other requirements is crucial to developing the full system. This can help show dependencies that may be needed for example Amazon Alexa may have some cloud or Wifi requirements for minimum acceptable performance.

Block Definition Diagram; Just like the small scale project a block definition diagram will be crucial to show the details of the system components and how other objects are related to the system. This gives a nice blue print of the system object structures.

Internal Block Diagram; For a medium sized project an internal block diagram will be essential to show a lot more details as there will be more complexity in the system. This will be very crucial for later implementation so you don't run into pit falls.

Sequence Diagram; At a medium sized project a sequence diagram can be very beneficial over an activity diagram since a bigger project involving embedded software will most likely have more back and forth data exchanges. This will can illustrated very easily and nicely with a sequence diagram.

F. A large-scale software embedded project such as smart refrigerator, smart car, Asimo humanoid robot

Requirement & Block Definition Diagrams; Similarly as the small and medium sized projects, a requirements and block definition diagrams are very essential. The requirements diagram only gets more important as the project gets bigger to ensure all requirements are known as well as the dependencies got the requirements. The block definition diagram on the other hadn't can get a little more abstract but still very important to show the actual structure of objects within the system to develop.

Package Diagram; For large scale projects the package diagram will be increasingly important. As projects get larger and more complex they will be harder keep organized. The package diagram will help with this by laying out the different packages or groups within the system and how they relate to each other. This is a very abstract view of the system organize into packages.

State Machine Diagram; This diagram shows the different states of the project machine driven by system behaviors or functionalities. This will be crucial for large scale systems like a smart car so that we know what we want the behavior of the system to be given different potential states.

2. (10 points) For each of the 6 challenges of digital engineering identified in EP-08, briefly explain potential solutions or techniques from computer science or software engineering perspectives to solve or mitigate these challenges.

A. Model Integration

One thing we can do to help mitigate the challenge of model integration is to keep terminology consistent between teams. For instance when talking about a user of the system as a 'Researcher' we should be sure to reference the user as a 'Researcher' in all models and no other label. Additionally we can help mitigate model integration faults by dedicating a lot of time to develop good quality requirements up front. If there are good requirements decided upon in the begging then once modelers go develop their respective models there will be less discrepancy between models. Moreover having good quality decomposition of the project will ensure model compatibility later on in the project life cycle.

B. Authoritative Data

One way we can help mitigate the the fault in identifying authoritative data is to do data analysis on our system. We can see where data is first written too and when it is updated. We should do some mapping of the data to document where the data originated and how the data updates over time. This will also imply some automated testing techniques as well in order test the flow of large groups of data to make sure we understand where fresh and stale data would be created.[3]

C. End-to-End Solution

In digital engineering End-to-End Solutions can have many challenges as some project require extended environments like cloud services or visualizer. The connectivity of an application is getting larger and larger. Because of this increase in connectivity it can be difficult to accommodate end-to-end solutions. A good way to meditate this is to slowly progress to an end-to-end solutions with continuous development and deployment. At first a project may have to rely on other services, it can slowly get to the point of supplying the entirety of the project scope. This DevOps approach can give good flexibility too as a project's services increase and can adapt to any changes. Additionally utilizing some good testing practices like alpha-beta testing can help get to end-to-end solutions through progress iterations.

D. Intellectual Property and Security Protection

To mitigate the challenges of Intellectual Property and Security Protection for digital engineering we can make sure to have strict access permissions to any data storages as well validating data source authenticity. This will be a huge help in preventing and security breaches and keep intellectual property safe. If the project has the resources then dedicating a team for security only would be a good idea.[4]

E. Workforce skills / training

To mitigate the challenges of workforce skills and training faults from digital engineering we can implement some paired programming techniques. This can be a great way for pairing one programmer with less experience with an expert programmer to get some development done while implementing some training at the same time.

3. (26 points) What are models (not only UML and SysML diagrams) that can be used to support the following software engineering activities? Provide supporting rationale to support your answer.

A. Project planning

Models: Incremental Commitment Spiral Model, Water Fall Model, Agile Programming, COCOMO Model, Gantt chart or Project Plan

Rational: For project planning there are a lot of models that can help support this process. There is of course the Incremental Commitment Spiral Model discussed in 577a. This model can help progress through a project while constantly analyzing risks and planning accordingly. There is also other model like water fall model and agile programming. Both have their different pros and cons but both are useful in project planning. Additionally also look at COCOMO model to address the cost estimation of project. Lastly a gantt chart or a project plan will be very crucial for supporting project planning activities.

B. Analysis of alternatives

Models: Incremental Commitment Spiral Model

Rational: The incremental commitment spiral model incorporate analysis of alternative during the valuation phase of the development. Here, the valuation phase will analyze the alternatives to how we can implement the project as well as conducting investment analysis.

C. Business projection

Models: Cost, Benefit, and ROI Analysis

Rational: Business projection can be analyzed by looking at cost, benefit, and ROI analysis. There are no specific universal models for these as they are usually looked at through line graphs showing the change over time for these measures.

D. Identification of use case or user scenarios

Models: Use Case diagram, Class diagram, Sequence diagram

Rational: For this activity we can utilize the Use Case diagram here. The Use Case diagram will allow us to capture all the functional requirements as well as highlighting any user scenarios in the process. A class diagram can be used here as well to show the details of the software construction which can help us see potential use cases and how they would be developed. Lastly a sequence diagram will be a nice complimentary diagram here to help us get into more detail about some of the user scenarios that we are planning to develop.

E. Requirements negotiation

Models: Win-Win negations, Requirements diagram

Rational: For requirements negotiations we can use win-win negations. This can help everyone come to an agreement between developers and the client where both parties win at the end of the software project. I would also suggest a requirements diagram. This diagram would help visualize the requirements in relation to each other which can help find dependencies that need to be considered.

F. Identification of logical architecture

Model: Component Diagram, Composite Diagram, Class Diagram, Object Diagram

Rational: For identifying logical architecture of a software system I think it would be very useful use a Component Diagram as it can show broad view of all the components and how they relate to each other. Additionally the Composite Diagram will give more detailed look into the components themselves giving more insight when trying to identify the architecture. Moreover a class diagram could be useful here as well depending on how low-level we want to get in which case the class diagram can give the structure of the objects that are in place and then we can use an Object Diagram for a specific instance to observe.

G. System performance analysis

Models: Level of service, Timing Diagram

Rational: For system performance analysis can be supported by using level of service and a timing diagram. The level of service will be very specific to the project at hand but some very broad levels of service may include transaction time, bounce rate, and usability of a system. A timing diagram can also support this by showing the specific timing constraints with certain software behaviors.

H. Risk analysis

Model: Value based risk ranking, Incremental Commitment Spiral model

Rational: A couple supporting models that can be used for risk analysis are value based risk ranking and using the Incremental commitment spiral mode. For Value based risk ranking the risks are calculated by getting the probability of loss times the size of loss that could occur. The higher the number the higher ranking a specific risk is. This gives a nice prioritized list of risks involved in the project. Additionally, we can use the Incremental commitment spiral model again here as this model emphasizes risk analysis at every increment, deciding to either move forward with the given risk or to mitigate it some how.

I. Technical debt management

Model: Incremental Spiral Model and Martin Fowler's technical debt quadrant categorization

Rational: There are a few models that can be used to support technical debt management. One we can again use the Incremental Spiral Model here as at every increment we are constant having commitment reviews and thus review any technical debt that we are still acquiring. We can also use the technical debt quadrant categorization by Martin Fowler. Here we categorize a technical debt by Deliberate or Inadvertent and Reckless or Prudent. This will give more details about our technical debt.[6]

J. Testing

Use Case diagram, Sequence Diagram

Rational: For testing I think it would be very useful to incorporate a Use Case diagram and a Sequence Diagram. In testing we need to make sure to get a good grasp of all the possible use cases and user scenarios to make sure we are getting good test coverage. To do this a use case can give us all the use cases that are developed and in an even more detailed view we can use a sequence diagram to get the exact process that we wish to test.

K. System deployment

Models: Deployment Diagram, Component Diagram

Rational: I would suggest using a deployment diagram for looking into the System deployment. The deployment diagram will give an illustration of all the abstract components in relation to each other and the deployment site. I would suggest using a component diagram here as well as it can give some insight into the overall structure of the project to see what components interacting with what other components.

L. System maintenance

Models: Component Diagram, Maintenance Diagram, Martin Fowler's technical debt quadrant categorization

Rational: When looking at the system maintenance procedures it can be useful to look at the component diagram, maintenance diagram, and the technical debt quadrant categorized by Martin Fowler. The component diagram can be useful so that you get an abstract overview of the main components we need to consider when thinking about maintenance. We should also look a the maintenance diagram which will illustrate the request for change of a model, highlight an issue, or to show a group of test that will be used. It will also be useful to look at the technical debt quadrant, this can help see what probably needs maintenance and can give some direction.[5]

M. System retirement

Models: Software Retirement Process Model, Component Diagram

Rational: For system retirement we should look at the component diagram again. I think this just like other activities that require an abstract review of the entire system can benefit from looking at component diagram. Additionally a Software Retirement Process Model was developed to help this activity. This model will illustrate the process of retirement from analysis to close down.[7]

References

[1] https://www.uml-diagrams.org/uml-25-diagrams.html

[2] https://sysml.org/sysml-faq/what-is-state-machine-diagram.html

[3] https://bnbuckler.github.io/ficam-identity/2_step-2/

[4] https://bnbuckler.github.io/ficam-identity/2_step-2/

[5] https://sparxsystems.com/enterprise_architect_user_guide/14.0/project_management/maintenance_diagram.html

[6] https://martinfowler.com/bliki/TechnicalDebtQuadrant.html

[7] Kajko-Mattsson, Mira, et al. "EM3: Software Retirement Process Model." (2014): 502-511.