Metode

Her præsenteres en objektorienteret analyse og design metode, der bygger på fire hovedgrundlag: Modellér IT-systemets omgivelser, fremhæv arkitekturen af systemet, genbrug mønstre der udtrykker velafprøvede designideer, og skræddersy metoden til opgaven og situationen.

Objekt: En enhed der har en identitet, adfærd og attributter.
Klasse: En samlet beskrivelse af objekter med samme struktur, adfærdsmønster og attributter.

System: Realisering af krav til model, funktioner, og grænseflader. (Læs: MVC designmønster)

De generelle fordele ved objektorienteret programmering er, at man kan modellere virkeligheden så det giver mening strukturelt og funktionelt.

Analyseobjekter: I analysen beskrives et objekts adfærd gennem de hændelser det udfører eller påføres. Fx. et kundeobjekt kan bestille og sende varer, to funktionaliteter der forekommer på bestemte tidspunkter og involverer en eller flere objekter og klasser. Beskriver fænomener uden for systemet såsom personer og ting der typisk er selvstændige i forhold til systemet. Ikke altid noget vi kan styre, men vi skal regisrtere de hændelser de udfører eller påføres.
Designobjekter: I designet beskrives et objekt derimod ved de operationer det kan udføre og dermed stiller til rådighed for andre objekter i systemet. Beksriver fænomener i systemet som er under vores kommando. Vi beskriver deres adfærd ved de operationer, som datamaskinen kan udføre.

Modellér omgivelserne

Problemområde: Den del af omgivelserne der administreres, overvåges eller styres af systemet
Anvendelsesområde: En organisation, der administrerer, overvåger, eller styrer et problemområde.

I et lønsystem omfatter anvendelsesområdet personalekontoret, mens problemområdet er de ansatte, deres overenskomster og arbejdstid.

I et lufttrafikstyringssystem er anvendelsesområdet luftlederens arbejde, og problemområdet er flyenes position, landingsbaner, terminaler, osv. Luftlederen, der skal fortolke systemet korrekt, har allerede en forståelse og et fagtermssprog for hvordan tingene skal fungere, og det er derfor ekstremt vigtigt at problemområdet er modelleret korrekt i forhold til brugernes forventninger. Det kræver derfor, at udviklerne studerer hvordan luftledere og piloter og andet personale skal interagere med systemet.

Fremhæv arkitekturen

System: En samling komponenter, der realiserer krav til model, funktioner og grænseflade.
Analyse og design skal føre til en overordnet forståelse af systemet.
Systemarkitekturen skal være let at forstå, fleksibel og brugbar.
Systemarkitekturen i denne metode beskrives simpelt som med tre komponenter: modelkomponentet, funktionskomponentet, og grænsefladekomponentet.
Hvis du er i tvivl om hvordan det her skal forstås er det præcis ligesom vi gjorde i vores OOP eksamensopgave - Stregsystem var modelkomponentet, StregsystemController var funktionskomponentet, og StregsystemUI var grænsefladekomponentet. Hvis du vil læse mere om denne model kaldes den MVC (Model View Controller model) på engelsk.

Genbrug mønstre

Det er altid en god idé, at anvende velkendte og allerede etablerede designmønstre i sit system. Dette kan være designmønstre som singleton, factory, osv. Dem lærer vi mere om senere. Designmønstre skal selvfølgelig omskrives så de passer til systemet, men at genbruge principperne i dem er god praksis.

Skræddersy metoden

OOA&D arbejder med fire hovedaktiviteter, som du kan læse om på side 12 og 13. Pointen ved dette underafsnit er, at der er ikke én rigtig måde at gøre tingene på, og der er ikke én rigtig rækkefølge at behandle de fire hovedaktiviteter i. Det er op til dig selv at vurdere hvilket perspektiv der er det vigtiste at arbejde med først.

Resultater

Det håndgribelige resultat af analyse og design er en dokumentation der beskriver hvordan systemet skal bygges af programmørerne.

Unified proces og notation

Som notationsmiddel bruges UML som vi er vant til fra P2. Unified Software Development Process er en proces ligesom top-down processen, hvor man i stedet analyserer, designer, programmerer og kvalitetssikrer dele af programmet i inkrementer.

Opgaver

1. Hvad er forskellen mellem objekter og klasser?

Et objekt er en enhed med identitet, adfærd og attributter. En klasse er en samlet beskrivelse af flere objekter med samme struktur, adfærdsmønster og attributter.

4. Hvorfor skal systemudviklere forstå omgivelserne for de systemer, de designer?

Når man forstår omgivelserne for de system man skal lave kan man derefter modellere det præcist og i et forståeligt forhold til virkeligheden. Dette gør implementation af vigtig funktionalitet lettere for systemudvikleren, når hans udgangspunkt for funktionalitet ligger i det fagtermssprog / funktionsflow der allerede er til stede.

5. Hvordan kan du skelne mellem et systems problemområde og anvdendelsesområde? Kan de delvist overlappe?

Problemområdet er de omgivelser, der skal styres, administreres eller overvåges af ens system. Anvendelsesområdet er den organisation der skal administrere, styre eller overvåge problemområdet.

Forelæsning

Representations always depend on your knowledge at the point of creating the representation.
The process of development circles between the user organisation and the development team. Once the problem is defined by the user organization, you can decide system specification, develop it and give it to the user organization.

OOA&A has a different approach. It focuses, rather, on what changes we want to do, based on the wishes and needs of the users. Identify their needs, the problem, and their visions.

There are two ways of thinking about system development. One is the world view where the system development is regarded as construction; rational, algorithmic. The other world view is that systems development is evolution, where competences and knowledge is richer than information and data. We have emotions, feelings, and experiences, and these things matter during development. Usually the development lead by the second world view is more dynamic in nature when compared to the construction.

Construction

Based on rational thinking. The steps are defind by 1. Analysis, 2. Decision , and 3. Action.

Construction is based on the idea of stepwise refinement of a solution to a given well defined and **stable** problem. We specifically assume that the problem does not change in nature.

The traditional approach to systems development in the contructionary sense is the waterfall approach.

Evolution

The evolutionary world view presents an iterative approach. You complete all the phases from analysis to testing, and then evaluate if the resulting system is what you want. The world view assumes that you learn things during development that you wouldn’t know before.

OOA&D

A method for analyzing and designing systems that:

1. Creates clarity about systems requirements
2. Establish a basis for implementation

The outcome is supposed to be
An analysis document: a description of system requirements.
And a design document: overview of the systems parts and their interactions.

We use UML as the notation tool.

Analysis

During analysis we are looking for phenomena in the context of the IT-system - in the real world. We are looking for identify objects, their identity and their behaviour.

Design

Here we are dealing with phenomena in the IT-system - not in the real world.

Construction, evolution and prototyping

The construction approach is a top-to-bottom approach that aims towards analyzing a specific problem formulation rationally and presenting a set requirement specifications from this analysis. Then the system will be developed based on these specifications. Very straightforward.

Another approach is the evolution approach. This approach is built more around the empirical mindset, as opposed to the rational mindset, where trial and error pave the way of progress during development. Instead of making requirement specifications, you then make a prototype of your system, typically a mock-up of a user interface. You then try it out with the future users of the system, and listen to their feedback. Then, you try another prototype built with the future users' feedback in mind. Iterate until a fitting prototype has been found, whereafter the system will be built.

The two approaches should not be regarded as alternatives to each other, but should always be sought to be combined instead. So where and when do we use the different approaches?

The construction approach is well suited to deal with non-changing, complex problems with low uncertainty involved. In practise, this could be:

"Build a program that finds the x'th fibonacci number, based on the fibonacci algorithm."

The problem is (sort of) complex, in that it involves a complex mathematical algorithm that has to work exactly as intended by its mathematical theory. It is also static/unchanging, in that the algorithm and the set of numbers we should be able to calculate never changes. The environment of the problem don't change. This also means that the uncertainty surrounding the environment of the problem are very low.

The evolution approach is well suited to deal with problems with high uncertainty and low complexity. For example:

"Design a user interface for a program that finds the x'th fibonacci number, to be used by IT-project managers."

Now, the problem has a human aspect, seeing that the future users are now a part of the problem formulation. With a human aspect like this, high uncertainty comes with it. There is no all-applicable method for designing a perfect UI for any program, which is why the evolution approach is well suited for tasks like these. Also note that the problem is not very complex; there are no complicated algorithms to take care of, not a lot of objects and relationships that need to be simulated. Just build a UI.

What the evolution approach does is that it makes a prototype of the UI for the program, and through feedback and new prototype iterations, it reaches a fitting prototype that is approved by the future users needs and wants. From there, the actual implementation begins. Naturally, this also means that it's difficult to set a definite timeframe and deadline for a project completely developed through the evolution method. This is why the construction and evolution approach are usually run in parallel during development.

TL;DR: Use the evolution approach for low complexity/high uncertainty problems, and the construction approach for high complexity/low uncertainty problems.

Experiments with prototypes

A prototype, as mentioned before, usually contains one or two of the main components in an OOA&D modeled system; i.e, one or two of the three user interface, function, and model components.

A prototype is usually limited in a set of different ways, be it functional, graphical or in terms of the modeled problem field.

Usually, a the experiments performed with these prototypes are split into two categories:
Exploring experiments uses a prototype to develop new design ideas.
Discussions may be around presentation prototypes or proper prototypes.
Evaluating experiments uses two or more prototypes to evaluate and choose a specific design. Prototypes proper help clarify this.
Evolutionary prototyping is a continuous process where the system is developed incrementally. In the end, the result is a pilot system.

No matter what experiment we're performing, the approach should always systematically follow these steps:

1. Planning
2. Development
3. Preparation
4. Field testing
5. Conclusion

Planning

Planning starts as soon as we realize that we need more information about the system. We need to define, in clear statements, what answers we need. For example "Can the user do X manually, or should X be automatized?", "Can we make X work?", or "Do the users want X?".
During the planning phase, the focus of the prototype also has to be defined. We do this by defining the focus in terms of the three main components (UI, Function, Model) and by asking ourselves the following questions: "What are we focusing on in this experiment?", "What are we purposely ignoring in this experiment?", and "What prerequisites do we have to fulfill?"

Development

The development works directly from the description of the prototype content. It's crucial that the development process is quickly finished, as it gets harder over time to maintain the end users' motivation to be a part of the development process.

Preparation

The field testing step of experimenting is the most crucial part, and it also has to be prepared for properly. Preparation includes determining how the field test should be carried out (should a user or the programmer use the program? Should it just be a discussion between developer and user?). It also prepares for the amount of realism of the test. It finally determines who to include and represent during the test (selection of represented userbase).

Field testing

This test is centered around testing the prototype in the chosen environment. You usually can’t interrupt and change a field test. It is also difficult to repeat a field test. Because of this, good preparation and documentation are absolutely crucial.

There are different ways to document the test. Getting the users to write a “diary”, for example from a set of checklists, is useful especially so when the developers themselves are the testing users. Not as useful with actual end-users as the testers.
Another way is to record the test with audio. Just remember to take notes along with the recording, as a test recording is a substantial amount of data.

Conclusion

Here, we need to conclude the results in terms of the question we started asking. The results will either be the inception of new experiments, or a sketch of a certain design. The results can be the building stones of a formulation and choice of system definition

TL;DR - fuck you read it

Lecture

Construction

- Bureaucratic approach to systems development
- The methods used remind us of mathematics; rational, logic
- Relies on overall plans
- User plays a passive role; they provide information, approve decisions
- It's a linear approach, no iterations
- Works well for **stable, unchanging,** and **well defined** problems; low uncertainty, high complexity
- Does not respond efficiently to change
- Closely related to the waterfall approach

Evolution

- Realizes that real world problems are rarely clear and precise
- Deals well high high uncertainty/low complexity
- Not suited for tight planning as it restricts possibilities
- Trial and error is a very important process in reducing the uncertainty
- Recognizes and emphasizes uncertainties
- The result is a *satisfactory* version of the system (**not** a perfect version)
- After this, the system may be developed further
- Deals with close communication with the users
- Works very well in changing environments
- Difficult to set a deadline
- Not a linear approach, iterations are key

There are several reasons for uncertainties to arise. One is simply the entrance of a human aspect in a problem. Another is simply misunderstanding the problem. Or misunderstanding how the application is going to be used.

During each iteration, keep in mind how it will affect the modeling of the problem domain (Model), the functionality of the system (Function) and how the User Interface works (Interface).

When experimenting with construction (requirement specfication) vs. evolution (prototypes) we saw the following advantages:

Construction

- More robust code and system
- Well written application

Evolution

- Better human-computer interface
- Quicker delivery of a functioning system
- Half the code

- Les robust code, less functional
- Less effort on planning, more effort on fixing and testing
- Lack of specifications

When choosing an approach, consider what factor (robustness, ease of use, etc) is important for the system to get right.

Construction is mechanistic in its world view, evolution is romantic.

Complexity: You have a lot of information, complex problem, but it's well defined.
Strategy of approach: Categorize and abstract, divide into sub-problems until each sub problem has a simple solution

This is construction

Uncertainty: You have insufficient or unreliable information about the problem and/or its solution. The requirements are unknown or partly described.
Strategy of approach: Collect more information about the problem and solutions, experiment with different solutions. Use information from experiments to iterate and improve until a satisfactory solution has been found.

This is evolution.

The principle of limited reduction: If you reduce the uncertainty, complexity is heightened, and vice versa.

Prototyping principles

Clients: initiates, makes agreements, signs contract
Users: work with or use the application system
Software manufacturer: develops and delivers the application system
Prototype: an early version of the system, is operational, is a communication medium between users and developers

Presentation prototype:

- Developed quickly using few resources
- Early
- Determination of overall requirements
- A first impression

Developed using very few resources. A typical presentation prototype is a paper prototype. It can also be made as a powerpoint with hotspots for mouse clicks. It contains no real important functionality.
On purpose is to see how intuitive the system is. This is why it's good to include the user in the process, but vocal support during the prototyping hinders our ability to see how intuitive it is.

Prototype proper:

- Provisional operational software system
- Illustrate specific functions
- Reveals design problems
- Realistic

Contains central functionality, as opposed to the presentation prototype.

Pilot system:

- Not the final system, but taken into use by the users
- Sometimes used in daily work
- Peak realism
- This is where we see if the system works in the application domain

Horizontal and vertical prototyping

In horizontal prototyping main focus is on the interfaces. You usually cover all of the interfaces, *some* of the functionality, but not yet concerned with the data models. Your focus is on the top layer of the system.

In vertical prototyping, a part of the system is implemented completely. This means some of the interfaces, some of the functionality, and some of the data modeling.

Principles

Specifying requirements is a difficult task.
A prototype is a part of the application system specification. When the user has said that a prototype is “what we want to go with”, that prototype becomes specification for what the system needs to look like/do.
Prototypes are a step on the way (increments) to the future system.
Prototypes are used exploratorily or to test solutions and ideas.

Cooperative interaction

Wizard of Oz prototyping means that **you** will be the computer. The user will use the interface, and you will simulate the reactions from the computer. You might choose to be in a different room, or to be present. It depends on the circumstances. This way, you can adjust in the moment, and analyse inputs. This is efficient, as it is easier to alter your response than to rewrite the code if something the user wanted to do didn’t work the way they wanted it to.

It allows you to do rapid iterative test of design ideas. It’s easy to make modifications to the prototype. It encourages collaboration and dialogue. It engages the user and makes it interesting.

It also has problems. The “wizard” can have incorrect behaviour. It is required that the wizard has a good understanding of the prototype and the problem, purpose of the system.
If the user becomes too aware of the wizard, it can hinder feedback.
There is also the concern of user bias, or emotional stakeholding; “this is not the way we do it!”

Exercises

Aalborg Lufthavn

Problemområdet:

Indebærer alle de fysiske elementer som fly, helikoptere, terminaler, gates, ruter, luftveje, tidsplan, vejr, grundkørselsfartøjer.

Anvendelsesområdet:
Flyvelederen, kontroltårnets personale

Flyvelederen skal have: Flyenes position i tredimensionelt rum, fart, rute, brændstof, liste over alle fly, direkte til flyene, tidsplanen, flytype, prioritering af fly.

Flyvelederen påvirker problemområdet ved fx. At ændre et flys rute, et ændre landingsbaners tilstand (åbne, lukkede)

System Choice

The system context

As we’ve gone over before, when designing a system we’re concerned with two main domains; the problem domain and the application domain. The problem domain is the part of a context that needs to be administered, monitored or controlled by a system, and the application domain is the organization that administers, monitors or controls a problem domain.
We use the domains to interpret a problematic domain, wherever that may be.

However, there are many ways to interpret the problematic situation. So how do we choose? We collaborate with the future users!

For reference, “System choice” is about defining the system we’re going to work with

System Choice: Three Sub Activities

The activity of defining a system is usually split into three subactivities:

Situation
This is where we aim to understand the situation and how stakeholders (the users) see it. A helpful tool for understanding a situation is to create an overview of the situation with “Rich Pictures”. Read about these in the book, they’re basically drawings that, for example, represent how the work flows in an office; they represent how the drawer sees the situation as it is. They give a clear vision of the inner workings of the problem and application domain and makes it easier for the systems designer to evaluate what changes need to be made.

Ideas
This is where we criticize existing traditions. It’s important to keep the “skeptic” mindset during this activity, as you need to question everything that is done traditionally, so that we can catalyze the process of thinking of new ideas and new ways to operate. This is usually also where new ideas are tested and tried using prototypes.

First, we look at existing solutions. You can visit other organizations or study standard systems on the market to get an idea of what the traditional systems are made of.

When looking at existing IT-solutions, answer the following questions:
- Which ideas lay the groundwork for the system?
- Do the ideas seem useful? Why?
- Will the ideas work in your context? Why?
- Can the ideas be adapted to your system? How?

You can also use metaphors to describe the system in a different way. This helps transfer ideas and experiences from one person with a set of skills to another person with another set of skills. For example, you can look at a library system as an inventory control system instead for the purpose of making your fellow developers understand it better.

For each metaphor:
- Create a list of aspects related to the metaphor
- Transfer these aspects to your target system
- Determine which of the aspects might be useful

When experimenting with prototypes, you should use the standard procedure when working with prototypes:

1. Planning
    a. Describe the prototype content
2. Development
    a. Start with simple prototypes on paper

    b. Then simple prototypes in for example PowerPoint
    c. Then functioning prototypes
3. Preparation
    a. How much should you cooperate with user during the prototyping?
    b. How realistic should the prototyping be simulated to be?
    c. Which users should be included in the prototyping?
4. Test
    a. Use the prototype
    b. Observe
    c. Document everything
5. Report results

If this procedure seems contentless, it's described in detail in the notes from an earlier course.

Systems
This is where we choose which IT-system we're going to develop.
The two proceeding sub activities aim to declare as many interpretations and possibilities as the situation allows for. What we now need to do is to concisely and accurately define these possibilities and the best way to do that is by authoring **systems definitions**.

A system definition describes a complete entity; it tries to gather and combine all different interpretations and perspectives into one concise and accurate description. It puts the focus on how everything should work together. It doesn't concern itself with details like how should the database be organized, but concerns itself with all-including properties like "User friendliness". User friendliness is only a relevant term for the system as a whole. No part of the system possesses this property on its own.
In the same way, a system definition also reflects certain boundaries that have been made.

See page 36 in the book for an example of a system definition.

The following are some practical tips for making a good system definition:

- Use general terms and focus on general properties
- Focus on ideas rather than the current situtation
- Make the definition concise and accurate
- Experiment with several system definitions
- Do not get caught in the mental tunnel-vision; think in new ways, criticize tradition

The FACTOR criteria

The FACTOR criteria (or BATOFF in the danish book) are a set of criteria (see the book) that any system definition should include. You can use the FACTOR-criteria in two ways. Either use it to support yourself during the making of a system definition, or make the system definition and then check if you satisfy all criteria.

Evaluating and Choosing a system

It’s not the developers job to choose the system. While the developer might be academically invested in choosing one system over the other and wants to argue this point, it is ultimately the users choice when choosing the system.

The choice of a system happens very early in the development process, and this can cause issues. During the process the developer may realize that changes need to be made, and this will require a re-negotiation of the system choice.

Knowledge domains in Systems Development

We need abstract knowledge to understand a domain in the context of the situation. We also need concrete experience to understand the user’s situation and discuss future changes.

Take a look at slides 8 to 10. They give a good and concise description of the different areas of knowledge and the tools and techniques you can use for acquiring abstract and concrete knowledge.

More on Rich Pictures

As said before, a rich picture represents how the illustrator understands the situation.
A rich picture either concerns itself with change or stability - how a situation should change, or how it currently is. It’s a great way to facilitate interactions between users and developers, and helps the developer organize their understanding and get a broad overview of the important aspects of the problem situation.

When drawing rich pictures, use the following:

Entities
This can be people or places. Remember that roles and tasks tie people together. More on this under Processes.

Processes
In a rich picture, you usually use arrows to describe information processing, work and production, planning, control, or development and organizational change.

Structure
In a rich picture, you usually use lines to describe a structural relationship between entities. This can be used to describe production and application, communication and agreements, ownership, membership and power relations all between different entities in the picture.

(Advice)
Rich pictures should generally contain much information and be open to interpretation by whoever is regarding it. It should coherently present processes and structures and show at least one problematic area. It should point to several computerized systems.

It should illuminate key aspects of a situation in a way that promotes the audience's understanding at many levels, and avoid representing data and data processing only (programmers tend to do this).
This is a lot of information, where the appended "rich" comes from, but it should only be rich of information, and not chaotic.

Modeling - Classes

When making paper prototypes, you're **supposed** to have some aural explanation with the video of the paper prototype walkthrough. It doesn't help much if you don't explain what you're doing while going through the prototype.

Model
A simplification of something in the real world. The purpose of the model is not necessarily to predict how it behaves, but also to understand it.

After creating a system definition, we need to get a deeper understanding of the problem we are trying to solve. The system definition is an important part of doing that. So are the users and people involved. This will be concerning the class activity.

Problem-domain analysis

The purpose is to identify and model a problem domain.

We already know what a problem domain is, so we will regard it as we are used to. We will regard the **model** as a description of classes, objects, structures, and behaviour in a problem domain.

We need to model the real world as *future* users will see it.
We also need to get an overview first, and then supply with details.

The output of this activity is a coherent model of a problem domain.

To do all this, we need to understand the users mental models. We need to create a tangible and visible model of these mental models. Because users are different, we need to include mental models and ideas, and systematically choose which one is the best fitting one.

Problem domain analysis is parted into three main activities:

To find relevant elements and events to model in our system.

To build the model and find the structural elements in our model; how do the different objects and events relate to each other?

Behaviour; how do the objects behave, and what triggers the events we have created?

When finding relevant elements to model, we need to consider whether or not they are static or not. A CPR-number is usually very static. Color can be static or non-static, depending on the context. Always understand the context before trying to model the environment.

Model the context

Problem domain: What are the important elements we need to model?
Application domain: What are the important parts of the context of the use we need to take care of?

Model of the problem domain

We need to design our model so that when something changes in the real world, that change should be reflected in our model. We need to be able to show change from the real world in our model to be able to show the same change in our application domain.This gives us an idea of what entities and events we need to model in order to be able to do this.

As mentioned before, we distinguish between three activities: Classes, structure, and behaviour. This will only be regarding classes.

During the class activity, we are looking for objects and events that are relevant to our system.
In the structure activity, we see how the classes and objects are conceptually linked together.
In the behaviour activity, we see which dynamic properties the objects have. What triggered the events, and what is the result?

Main concepts when analyzing the problem domain:
Objects are entities with identity, state and behavior.
Classes are descriptions of collections of objects sharing structure behavioral pattern, and attributes.
Events are instantaneous incidents involving one or more objects.

Main principles when analyzing the problem domain:
Classify objects by classifying them in the problem domain. We then characterize the objects **through their events**.
We need to have an open mind when coming up with candidates, but also have to select critically thereafter.

The result should be an event table with classes and relevant events.

Classify objects and events

We try to identify different objects that share the important characteristics and group them together in classes. Similarly with events, there are a lot of different events happening. However, some of these events are expressions of the same thing. That is, we generalize and make abstractions.

Examples of classes:
Customer
Assistant
Agreement

Example of events:

Agreement made
Customer serviced

See slide 12 for an example of an event table. You can see the different classes and what events are tied to them. That is, their behaviour.
When a customer is reserved, it's the assistant's job to make the reservation. The result of that event is an appointment, which is why that is also included.

It's important to remember that the purpose of this activity is to get an understanding of the problem domain that allows of to make a model of it.

Activities in “classes”

Find candidates for classes: Find potential classes that could be included in the system. Be open minded.
Find candidates for events, same thing.
Then we evaluate and select systematically. The selection is based on the criteria described in the book, but they are pretty intuitive on their own.

Find classes

Make a list of all potentially relevant classes. Consider a lot of sources like your own perception, existing descriptions and like rich pictures and system definitions and stuff like that. Or, you could collaborate with prospective users.

The *names* of the classes must be **simple and reachable, originate in the problem domain** and should be able to **describe a single instance** of that class**.**

Based on the first list of potential classes, we can start reducing it.

Generate a similar list of candidates for events.
Here, we can use our own perception or existing descriptions as sources.
Eliminate verbs related to the way users carry out their job, because that belongs to the application domain. The naming conventions look a lot like the ones for classes.

Evaluate and select systematically

General evaluation criteria
Is the class or event within the system definition?
Is the class or event relevant for the problem-domain model?

If either is a no, discard the candidate.

Evaluation criteria for classes

Can you identify objects from the class? If you can't you probably don't need the class.
Does the class contain unique information? If it doesn't, you are at the risk of creating redundancy in your system. If your class contains the same information as another class, two classes need to change at some event, which creates a lot of overhead.
Does the class include multiple objects? Unless we can identify multiple instances of this class, it might not be necessary to model it. Classes are a collection of shared properties and behaviors across objects. Therefore, if there is only one instance of it necessary in the system, the abstraction level is nowhere to be found. Remember that this is USUALLY how it's done. A boss, for example, can be modeled because it can handle events that customer classes cannot, even though there is only one boss.
Does the class have a suitable and manageable number of events? If your class has too many events, you might need to split that class into different subclasses.

Side note: Usually, we don't want to have classes that are just lists of other classes in our system. For example, a list of customers in a banking system should not be a valid candidate, because there is no abstraction level. However, if the list of customers is an explicit part of the problem domain (fx. Mentioned in the system definition), it should be included as a candidate regardless, even though it doesn't satisfy some of the criteria.
As an example, a system definition for a mobile phone system may mention that the system must contain a contact list, and that this must be updateable. This means that the class "Contact List" should be a central class, even though it's just a list of the class "Contact".

Evaluation criteria for events

Is the event instantaneous? It can be hard to determine when a continuous activity happens. The event must be instantaneous.
Is the event atomic? Can it be divided into subevents? If we can, we need to do that. An event must not be atomic.
Can the event be identified when it occurs? If we can't, we need to look for other events.

Look at slide 22 for the selection of different class candidates.

We don’t need a customer database, as we can get this from just grouping customers. We don’t need a separate class for that. Same thing for appointment book.
We don’t need a cash register class, as it is not a part of the system definition. Treatment performed is an event, not a class. Desired vacation should not be a class since it is not a part of the system description. The work schedule would be covered by the plan.
Boss, assistant and receptionists need to be there.
The system definition says that every employee has their own chair, which is why the chair class becomes redundant.
Look at slide 23 for the selection of different event candidates.

We probably don’t need customer arrived, since it is pretty evident if the customer is there or not.

Note the “agree” that is a part of the events. This is a good example of bad naming. We are talking about a specific type of agreement, and that is not clear from the naming of the event.

Lecture 4 exercises

System definitions for the airport traffic control system

We need to make sure that our system definitions satisfy all the criteria in the FACTOR (or BATOFF) criteria set.

Notes from Esben: *In any system, the system itself is actually part of the application domain. As long as something can affect the use situation, it should probably be part of the application domain. However, it’s up to you how much detail you want to get into, and whether or not the repair man should be included in the application domain because when he repairs the system it goes down, and that affects the traffic control.*

S1

Functionality: The system should allow the user to communicate with all vehicles, allow take-off and landing, and supply air vehicles with routes.
Application domain: The system should be designed for use by air traffic generals relevant personnel in the radio tower.
Conditions: The system should not allow for downtime or erroneous information. The system should be robust, safe, and completely tested. The system cannot undergo live-testing.
Technology: Should be developed for a Windows PC. Development must include interfacing for radar and whatever communication hardware they currently apply for radio tower/airplane communication.
Objects: The system should model all air and ground vehicles and the map of roads,

including runways and taxi-roads. Furthermore, the system should contain a timed schedule for when planes should arrive and leave the airport.
Responsibility: Flight monitoring and communication medium.

The system should be responsible for monitoring and facilitating communication between radio towers and planes. It should be designed for use by several air traffic controllers and other relevant personnel in the radio tower. The system should allow the user to communicate with all vehicles, change the status of roads, allow for take-off and landing, and provide active airplanes with routes. Following this, the system should contain information about all vehicles, road maps, timed schedules, and airspace. The system should not allow for downtime or erroneous information. The system should be robust, safe, and completely tested. During development, the system cannot undergo live-testing.

S2

Functionality: Warn when a collision course is entered, divert from course, communicate with other planes,
Application domain: Pilots
Conditions: The system should not allow for downtime or erroneous information. The system should be robust, safe, and completely tested. The system cannot undergo live-testing.
Technology: Whatever hardware airplanes contain
Objects: All planes in the same airspace and itself. Positions, current routes, velocity, size
Responsibility: Collision prevention

The system should automatically handle collision detection and prevention in airplanes. The system should be designed for use by pilots, and should be developed for implementation in the proprietary airplane hardware. The system should allow for the pilot to communicate with other planes, as well as alarming the pilot when a collision course is entered. The system must be able to represent information regarding the position, course, velocity, and size of itself and all other planes in the same airspace. The system should not allow for downtime or erroneous information. The system should be robust, safe, and completely tested. During development, the system cannot undergo live-testing.

S3

Functionality: The system should automatically communicate with all vehicles, allow take-off and landing, and supply air vehicles with routes based on information received from all vehicles and road maps.
Application domain: Sys_admin, root, Zezima, ReturnOfWilderness, Pilots
Conditions: The system should not allow for downtime or erroneous information. The system should be robust, safe, and completely tested. The system cannot undergo live-testing.
Technology: Hardware with the ability to receive information from all airplanes, a central server, and a visual interface.
Objects: The system should model all air and ground vehicles and the map of roads, including runways and taxi-roads. Furthermore, the system should contain a timed schedule for when planes should arrive and leave the airport.

Responsibility: Monitoring flight control as well as making decisions regarding routing and permissions, and communicating these decisions to the pilots.

The system should monitor flights and make decisions based on the best possible routings and communicate these decisions to the pilots. The system will be in contact with all vehicles and convey information to the pilots regarding landing and takeoff as well as other relevant information. The system will be monitored by a system admin as well as a flight controller. The pilots themselves will be in contact with the system, and can impact the system by not following the instructions given.

Classes

Profile
Component
Packaged Solution
Energy Label

Events

ProfileCreated
ComponentCreated
PackagedSolutionCreated
ComponentAddedToPackagedSolution
ComponentRemovedFromPackagedSolution
EnergyLabelCalculated
PDFCreated

	Profile	Component	Packaged Solution	Energy Label	Database
Profile Created	X				
Component Created	X	X			X
Packaged Solution Created	X		X		X
Component Added To Packaged Solution		X	X		
Component Removed From Packaged		X	X		

Solution					
Energy Label Calculated			X	X	
Energy Label Formatted			X	X	
Energy Label Exported	X		X	X	

Modeling - Structure

Until now we've looked at tools to find classes and events from a deciphered problem and application domain.

This is going to be regarding how the classes and events are related to each other.

See Slide 3 for an example of an event table for last exercise modules.

Activities in "Analysis of problem domain"

See Slide 5 to see where we are in the process.

First, we look for candidates for structure. We study 1. abstract and static relations, 2. concrete and dynamic relations between classes and objects. We identify generalizations, aggregations, associations, and clusters.

Thereafter, we explore what patterns we can apply to our structure.

Then we evaluate whether the structure is used correctly. The structure should be conceptually true. It should also be as simple as possible. This is because it is supposed to supply us with a UML diagram, which can become very messy very quickly.

Class structures and object structures

Class: static, conceptual relations between classes. These are relations that never change. They concern all the objects of the classes.

Object: Relations that can be dynamic and changing. Some objects can be in such relations, while other objects from the same class may not.

We will look at the two relations: aggregations and associations.

Generalization

Pretty much a “is a” relation. For example, a taxi **is a** passenger car. A maths teacher **is a** teacher. See slide 6/9 for an example of the generalization in UML and why you want to keep it simple. This is usually where you begin to think about inheritance in object oriented programming.
Note: if a class in the UML diagram is written in *italic,* then it is an abstract class.

See slide 10 for the vent table for the hair saloon. We readily see classes that can be generalized, like apprectince and assistant into an employee class.

Cluster

We group classes that are conceptually tied together in what looks like a folder structure. See slide 12. Note that there is no line between owner and clerk, but they are clustered together because they are conceptually related.

Aggregation

Usually annotates one of three alternatives:
Whole - part, like “the car is made up of these parts”
Container - contents - The main class is the container, the aggregated classes are the contents.
Union - member - a bit like container-contents, but it makes a bigger distinction when the contents are too few and far between.

The 1’s and 2’s in the diagram mean this: A cylinder can only be on one car, but a car can have two or more cylinders. An engine can be in one car, and a car can only have one engine.

Association

A “knows” or “associated-with” relation.

A person can have a relationship to a car, or not have one at all. This is why the multiplicity 0..* is at the car. A car has at least one owner, but can have more, which is why multiplicity 1..* is at the person. Slide 14.

See slide 15 for more candidates for relationships. Customer and Appointment share the same events, so we can assume that they have a relationship in some way. The relations should be an association between the two. Furthermore, an appointment can only have one customer associated with it.

Patterns

Role pattern

See slide 17. A person can have roles, which is annotated by a mix of aggregation and associations. A person can have several roles, and a role can only be aggregated with one person.

Relation pattern

The top right relation doesn't say much about the relation between the two classes.

Instead, we create a class that annotates the relation between the two classes or objects. It's important to note that in the example, the person aggregates (i.e. owns) the ownership, the car does not.

Hierarchy pattern

We can have multiple layers where each layer organizes the layer below. The structure should be self explanatory, see slide 19.

Descriptor-item

In a library, they will probably have several copies of the same book, but don't want a unique description of all the copies, because it's the same book. The slide 20.

Evaluate

Once we have our candidates for relations, we need to see if they satisfy some criteria. First and foremost, the relations must be used correctly.
They must be conceptually right, names, concepts and structures should correspond to the users understanding.
The structure must be simple, avoid unnecessary generalizations and aggregations. Check against the system definition.

It's not always obvious what kind of relation we should choose. For example, an argument could probably be made for all the structures on slide 23, but it should probably be association because of the nature of university lectures.

When we consider relations:
Can objects exist independently of each other? Are they equally ranked and can the connection be changed for one pair of objects to another? If you answer yes to two or three of these, then it's probably association. Otherwise, it's probably aggregation.

See slide 27 for the rest of the relations between the classes. See also that the “Plan” class was removed and replaced with four now classes, in effort to employ the patterns.

Note that in the last slide of the example, appointment is associated with Work because when an appointment is made, the day schedule changes because a new Work time period has been created.

When we have a diagram, we evaluate it against our event table. Do the relations between our classes in the event table correspond to our newly constructed UML diagram?

To round up: **Class structures** include **generalizations** and **clusters. Object structures include aggregations** and **associations**.

Modeling - Behaviour

This is the last activity in the problem domain analysis
The classes and methods are required to find the behavioral patterns we need.
We previously went over the elevator example when trying to construct an event table.

The activities consist of:
- What are the dynamic properties of objects?
- Event trace, behavioral pattern, and attribute.
    - Event tracing is a sequence of events involving a specific object.
    - A behavioral pattern is a description of possible event traces for all objects in a class.
    - Then we find the attributes, the descriptive properties of a class or an event

The subactivities:
1. **Describing behavioral patterns** - event trace, generate general patterns, study common events (events that several objects take part in)
   We also try to make our behavioral models show the legal behaviour of an object, because then we also show what behaviour is ILLEGAL.
2. **Explore patterns** - We’ll be looking at three different patterns - the stepwise relation pattern, the stepwise role pattern, and composite pattern.
3. **Consider structures and classes** - Study structure and behaviour, remember that behaviour can be inherited.
4. **Describe attributes -** Derive class attributes from behavioral patterns.

If we have a bank account, a general thing we always look for are the events that create our objects and the events that terminate it. We always have to find out when the object starts existing in our system and ends. In a bank system, a customer starts existing when they open an account. Based on that event, we get a *state* of the object. Look at slide 8 for a diagram describing the state of the customer. Amounts can be deposited and withdrawn, but those events don’t change the event. This is an unstructured state chart diagram.

Slide 9 is a more structured state chart diagram. This is an example of an author that starts existing when a paper is registered. Before anything else can happen, the paper must be submitted. After submitting, there are no more events that involve the author.

The main difference between the structured and unstructured diagram is that the structured diagram is a sequence, where the unstructured one is only sequentialized by two very special events, and the rest do not change the state of the object or come before or after each other.

The reason behind this distinction is that structured behavioral patterns are very important, and if we have an unstructured behavioral pattern, we need to look for structures in it.

Look at slide 10 for statechart diagram notation. It is rather self explanatory.
Sometimes, an object can toggle between two states, which is shown in the bottom right corner of slide 10. This is called “indirect iteration”.

Find behavioral patterns from event traces

For each class:

- Which events cause the creation of a problem domain object?
- Which events cause the death of a problem domain object?
    - When an object is “dead”, it just means that it cannot be involved in any more events.

Typical events traces

- Which events occur in a sequence?
- Is the overall form structured or unstructured?
- Are there any alternative events?
- Can a given event occur more than once?

Example

We return to the hair saloon problem. A customer starts existing in our problem domain when they first make an appointment. After this, they can cancel reservations, make new reservations, and get treated. During this, they are considered active. We have also identified three attributes; name, address, and phone number.

The appointment object starts existing when a reservation is made by a customer. After activation, the state of the appointment is “planned”. The appointment can be cancelled or it can be “treated”, both leading out of the “planned” state.

Hierarchical states

This is where you have multiple events and states within an overall state. For instance, if we have identified the sequence of events that leads to different states, but during each of these states, the object can be terminated, then we have to option of creating an overall state and let the termination arrows go from the overall state to the termination state. See slide 14 for an example. It’s the bottom example, which is easier to read than the top one.

Look at the apprentice statechart. An apprentice starts existing when they are employed, and during this time, they can always be resigned, and thereby terminated.

Inheritance of behavioral patterns

When we have different specializations of a superclass, we can identify behavioral patterns that can be inherited by all subclasses. See slide 16. The superclass has a behavioral pattern that is inherited by the subclass, but is also extended with another behavioral pattern. Also, see slide 18 for another example of behavior inheritance.

Stepwise relation pattern

An object cannot enter a relation to objects of another class until after another relation of higher hierarchy is already established. The stepwise relation pattern is only concerned with objects in a hierarchy. For example, a student must be assigned to a semester before being assigned to a class. There is a specific sequence in which the relations can be made.

Stepwise role pattern

Remember how we modeled roles when objects could dynamically change between different roles. In this pattern, we have a defined sequence in which the different roles can be entered. For example, a sale can be either an offer, and order, or a delivery. However, these roles must happen in a specific sequence.

Composite pattern

Useful for describing the creation or destruction of a hierarchical structure that is unknown at the time of model development. The general abstraction is that we have an aggregation of something like “a car engine aggregates a lot of different parts”. We don’t know the specifics. We can model it by saying a part is either a simple part, or a composite of other parts. For simple parts, once we have it, it’s ready. If it’s not, then it’s a composite, then we need to start assembling it (state: assembly), and when the part is readily put together, it can be mounted in the overall object.

In short, for different parts, they will always have a “ready” state, what matters is how and when we get to that state depending on whether the part is a simple one or a composite.

Evaluate classes and structure

When we have the behavioral patterns, we evaluate the classes and structures. We look for possible generalizations that we didn't see. If the same event is tied to two classes, we can consider that one class is a generalization of the other class. At object level, we can look at two objects having common events. Also, if a class aggravates another class, they should share at least one common event.

For instance, the bank customer was a little bit confusing because when an account opened, the account state was open, and the withdraw and deposit were iterative events that didn't change the class immediately. We can change the class to help us make sense of the diagram. See slide 23 for an example.
Return to the hair saloon, and look at the plan class.
Every day, he creates a daily schedule. It starts existing when it's scheduled. There is only one way out of the "planned" state, and that is to agree the schedule. During the "agreed" state, several iterative events can occur.

Remember that when we made event tables last, we used checkmarks. Now, we know a bit more, whether an event can happen once or multiple times. The plus signs indicate a single occurrence, and a star indicates several occurrences.

Describe attributes

We distinguish between three types:

- Attributes connected to events
- General information about the objects in the classes (name, address)
- Attributes that we can derive from other attributes.

For classes:

- What are the general characteristics of the class?
- How is the class described in the problem domain?
- What basic data must be captured about objects from this class? This can be both static and dynamic objects. If objects are state-changing, we might need to capture information about this.
- What results from an event trace must be captured?

For events:

- What time did the event occur?
- Which amount did it concern?

Result

The result of a behavioral pattern analysis is a behavioral pattern with attributes for every class in a class diagram.

Exercises

5.15
Legal event trace:

- Account is created on XX/XX/XX. 100 dollars are deposited. 50 dollars are withdrawn. Account is closed.

Illegal event trace:

- Account is created on XX/XX/XX. 100 dollars are deposited. Account is closed. 50 dollars are withdrawn.

5.17

Stepwise relation pattern

A student must have a relation to a semester before having a relation to class in that semester. Thereafter the student can have a relation to a group in that class.

A car mechanic must have a relation to a car before generating relations with the components that are faulty. After this, he can generate a relation to the specific part that is faulty. A car aggravates components that aggravate parts.

Stepwise role pattern

Noter:

Attributter skal IKKE referere til objekter.

Application-domain analysis - Use

Until now, he hasn't started giving feedback on our assignments.
Keep an eye on moodle, as the next assignment will become public during it.

The assignment

We had a system definition and an interview.
There is an example of an event table in the first slides.
We need the last "Price group created/cancelled" and "station created/closed" because while they're not a direct part of the system definition, they are events regarding objects we need to monitor and administrate.
See that Esben has utilized both the hierarchy pattern (Station, price group, car) and the role pattern (Contract, reservation, rental).
Note that the Customer class should be an abstract class.
Question: Could you argue that price group could be an attribute?

Answer: Maybe, when implementing, we could do that. But right now, we are trying to model it correctly to get a good understanding of the problem domain. The price group is a pretty big part of the system definition and problem domain, so we need to include it.

Application-domain analysis

Model the context

The context is the problem domain and application domain. Remember that the application domain is the organization that administrates, monitors or controls a problem domain.

Emphasize the architecture

When we did the problem domain analysis, we gained a lot information about the model part of the MCV system. Now, we're going to learn a lot about the interface components.

Application-domain analysis - result

We're going to talk about use today, functions later, and interfaces in the end. These are the three activities in the application-domain analysis.

So, we're going to look at how the system interacts with people and systems in the context. The divisions between problem and application domain is that it gives us insight into some other aspects than problem domain would. The model resulting from the problem domain analysis is likely to not change a lot. Interface properties, however, which is what we're going to find here, is going to be subject to change a lot more than the model. Furthermore, changes in the interface do not require changes in the model, while the vice versa situation isn't always true.

Purpose of application-domain

To determine a systems usage requirements.
The output should be a complete list of the systems overall usage requirements.

Use

The purpose is to determine how actors will interact with a system.
An actor is an abstraction of users or other systems that interact with ours.
A use case is a pattern for interaction between the system and actors in the application domain.
We determine the application domain with the use cases, and evaluate use cases in collaboration with users.

The outcome should be descriptions of all use cases.

Result(1): Actor and use case overview

The actor table. Looks like an event table, but is different.
The main difference is that the elements in the column to the left are no longer events.

Result(2): Actor and use case specifications

A description of the actor (see the slide), and a concise description of a use case relevant to the actor.

Some of the activites we're going to do here is to find actors and usecases, and we do this by analyzing tasks. We then identify the actors, describe them, same thing with the use cases. Then we structure the actors and use cases with each other. We then explore for patterns, and then evaluate the actors and use cases for consistency (are they related to the same task? Logically sound?). We can also use prototypes to evaluate our actors and use cases.

Analyzing tasks

We determine the application-domain with use cases. We avoid detailed descriptions of "how it used to be".

The goal is to get an overview of the work tasks in the application domain, and especially the division of work and the task boundaries.

What are our information sources? Mostly the system definition, we have some rules and procedures from textbooks on how to do things, we can observe people, interview, participate in the work with users, and so on.

Example

When describing tasks, give them:

- Name and contents
- Purpose
- How are the tasks assigned?
- Who performs the task?
- Relation to other tasks
- Result

See the slide for an example, I'm not fast enough for this.

Find and describe actors

When identifying actors, we determine the division of work and the task related roles in the target systems context. Different users have different roles, so our users/actors do not relate to the single individual, but more the role itself.

When describing actors, give them a goal, characteristics (aspects of the actors use of the system) and an example (general characteristics).

See the slide for an example of an actor description. Those are examples of two very different actors that are both interacting with the system, but both in the role of the account owner.

Identify use cases

When we identify use cases we describe how actos interact with the system to complete their tasks. We try to minimize the overlap between use cases. We also try to make them coherent, so that the actor relates to the use case.

We write scenarios with examples of how to complete the tasks. We do this with text and state chart diagrams.

See the slide "Describe use cases" for two examples of how to describe a use case.
You see both a description in the form of a textual scenario, and a state-chart diagram.
In addition to the text, we can add the objects and functions required to support the use case. Note that the words at the arrows between states are not events, but interactions. For example "prompt for code" is an interaction between actor and system, not an immediate event.

Example: Bank

Another example, not the ATM system, but the system with account owners, and so on. Note that "liquidity monitors" are not people, but is another system that interacts with this system.

We can also relate use cases to the actos graphically in the slide "Use case overview(1): Use case diagram". It should be self-explanatory. Another way of illustrating this is to make an actor table like the one we saw before. We have a list of actors and a list of use cases and the checkmarks that associate the actors with the use cases.
We can also cluster the use cases within an overall focus and model which of the actors are related to the cluster(s) or to single use cases within the clusters.

Patterns

We have patterns that we try to reuse when looking at use cases.

The procedural pattern
We have some states that can only be entered given that some interactions are executed in a specific procedure. This pattern is useful when we want to ensure that business rules are being followed. That can be very efficient.

The material pattern
Here, the actor can do almost everything in any order.
The generic way of showing this is you have one general state, and iterative actions on that state. We can also have indirect iterations where one action can lead to a different state, and return to the general state when another action sends it back.

Similarities and differences between use cases, actors, events and objects

See the slide for a comprehensive comparison.

Actors and classes describe the more static attributes of the domain we're trying to model. Use cases and events are more dynamic in nature, and describing the dynamics in the domain we're trying to model.

Evaluate systematically

Each use case should be simple and constitute a coherent whole.
Description of actors and use cases should provide understanding and overview of the application domain.
Use cases should be described in enough detail to enable identification of functions and interface elements.
Test patterns of use with users.

We mentioned that prototypes can be used to test use cases. See the slide for the different acitives when testing use cases with prototypes.

Evaluate social changes

What are the changes in the application domain when our system takes action?
When we are trying to identify social changes, we can look at work content, autonomy and control, social relations, and education and development.
In the mechanistic extreme, the work content may be specialized jobs, and many things that are being regulated. Autonomy might be monitoring, the stressful load of employees, and so

on. The social relations may include less security, the users may become alienated from the system, not a lot of social interaction and so on.

The romantic extreme differs.
In the work content, consequences might by no division of labor, no procedures and rules, and everything is regulated by consequences.
This is going way too fast to write down, look at the slide.

It’s important to understand this part of the evaluation, as it’s pretty important to understand the social changes your model of the application domain might lead to.

Exercises

Se SW3_Project_Temp.pdf der ligger i denne lektions mappe. Se afsnittet “Application Domain” for at se hvordan vi har lavet vores application domain analysis.

Opgave 15

Aktør: Mobilhaver

Formål: At kommunikere med venner, familie og bekendte vba. sin mobiltelefon når face-to-face kommunikation ikke kan lade sig gøre.
Karakteristika: Systemet omfatter mange forskellige brugere med unik information og varierende teknisk erfaring.

Eksempler:

1. Mobilbruger A har altid haft problemer med at huske navne og telefonnumre. Han har ikke meget forstand på teknologi, og har behov for at interaktion mellem ham og telefonen er så begrænset som muligt, da han helst foretrækker face-to-face kommunikation.
2. Mobilbruger B er teknisk interesseret i al teknik hun rører ved, og prøver derfor sin telefon af for funktioner og muligheder. Mobilbruger B agter at bruge sin telefon så meget som muligt, og prøver for det meste at kommunikere igennem denne frem for face-to-face kommunikation.

Brugsmønstre:

1. Anne-Lotte skal ringe til hendes mor. Hun kan desværre ikke huske hendes mors nummer, og kan heller ikke huske om hun har gemt det tidligere. Hun prøver at søge efter sin mors navn i kontaktbogen, og finder en gemt kontakt under hendes mors navn.

2. Hans-Erik snakker sammen med en kollega efter arbejdet. Hans-Erik har altid været dårlig til at huske navne, og dermed også telefonnumre. De aftaler at mødes senere den weekend, og for ikke at brænde sin kollega af på aftalen senere gemmer Hans-Erik kollegaens nummer i sin kontaktbog på telefonen.
3. Torben-Arkibal er blevet ringet op af en gammel ven for to dage siden. Han glemte at fortælle ham noget vigtigt dengang, og har brug for at få fat på ham igen. Han kan desværre ikke huske hans nummer, da det var hans ven der ringede til ham. Torben-Arkibal kan dog huske hvornår de ca. snakkede sammen. Han går ind i sin samtalelog og finder deres samtale, hvori han finder sin vens nummer.

Aktørtabel

Da vi kun har én aktør bliver de bare en liste af brugsmønstre der er koblet til aktøren.

Mobilhaver
- Ring
- Læg på
- Tag telefon
- Opret kontakt
- Slet kontakt
- Find kontakt
- Find samtale

Opgave 6.17

Aktører

Passager

Formål: At komme fra en etage til en anden etage vba. elevatoren.

Karakteristika: Mange forskellige personer der skal forskellige steder hen, måske med varierende teknisk erfaring.

Eksempel: Passager A er en gammel dame der ikke har meget forstand på noget teknologi. Behøver et intuitivt og simpelt interaktionssystem.

Passager B har meget travlt når han skal frem og tilbage i bygningen, og kræver derfor en hurtig respons i systemet.

Elevator controller

Formål: At sende elevatoren til den forespurgte etage.

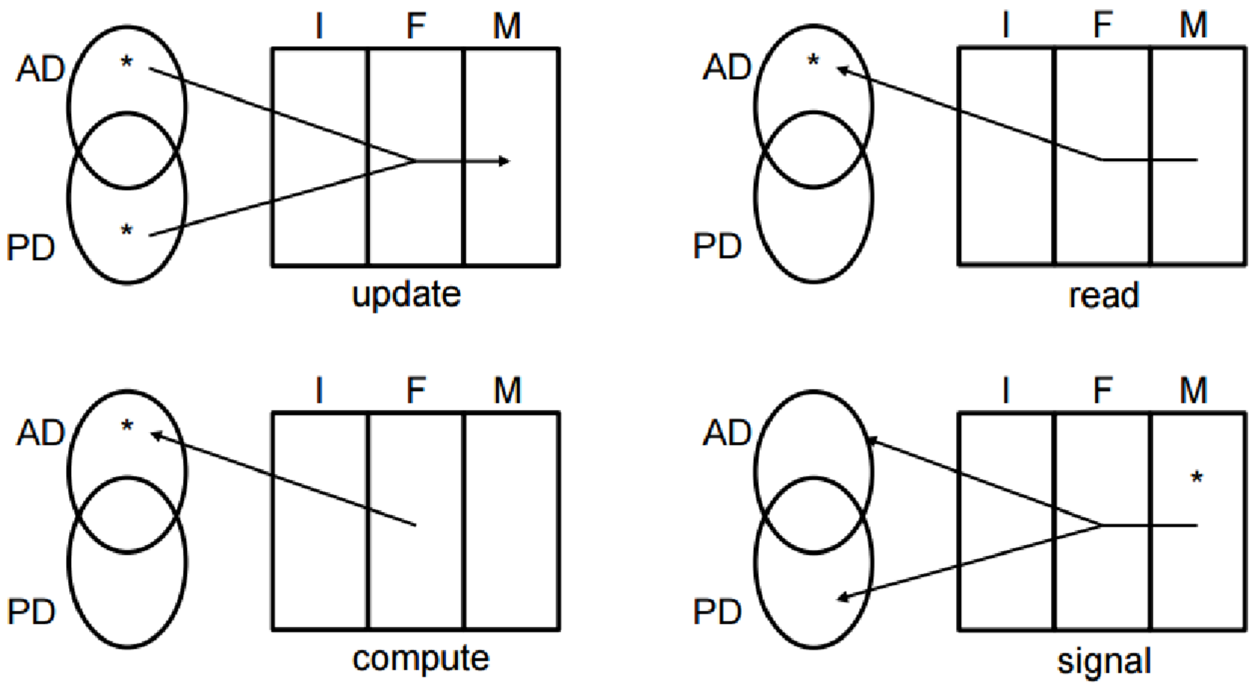
Karakteristika: Der er én af dem i hvert elevator, og interageres med via et panel af knapper.

Eksempel: Controller A er en computer der sidder i elevatoren, og modtager passagerernes input. Controlleren udfører derefter den logik der skal til for at få elevatoren fra A til B.

Funktioner

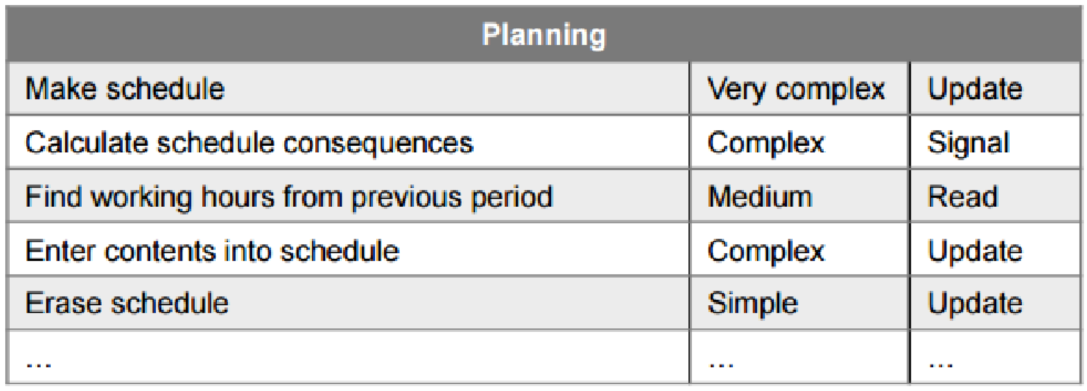
(Jeg var på Sjælland i en uge da den her forelæsning var der, så jeg "ansatte" et gruppemedlem til at tage noterne for mig. Det er derfor de er på dansk, og skrevet på en anden måde end normalt, men han har forsikret mig, at han har fået alt væsentligt med fra både bogen og forelæsningen!)

-
- Funktioner har til formål at fastlægge krav til informationsbehandling i anvendelsesområdet.
- En funktion er defineret således: En facilitet der gør en model anvendelig for aktører.
- Der er fire generelle funktionstyper:
  - Opdatering
    - Aktiveres af en hændelse i problemområdet og resulterer i tilstandsskift i modellen.
  - Signalering
    - Aktiveres af en ændret tilstand i modellen og resulterer i en fremvisning over for aktørerne i anvendelsesområdet eller et direkte indgreb i problemområdet.
  - Aflæsning
    - Aktiveres af en aktør i applikationsområdet og fremviser relevante dele af modellen.
  - Beregning
    - Aktiveres af et informationsbehov i en aktørs arbejdsopgaver og består i en beregning, som involverer information fra aktøren og modellen, hvilket resulterer i en fremvisning af beregningens resultat.
- En grafisk afbildning af de fire funktionstyper ser ud således, hvor pilen indikerer effekten af udførslen og stjernen (*) indikerer initiativet til udførslen. F.eks. kan en opdatering modtage data direkte fra problemdomænet (PD) eller data der indtastes af en aktør i applikationsdomænet (AD).
- Ved funktionsanalyse ønskes det at alle funktioner identificeres, komplekse funktioner specificeres og en komplet liste af de specificerede funktioner udarbejdes.



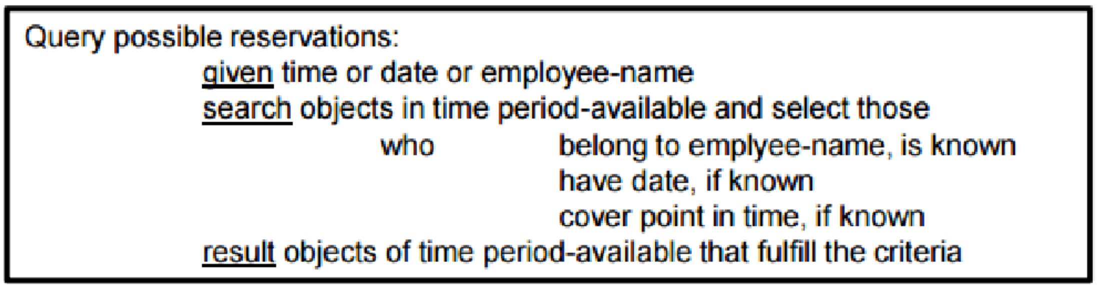
Eksempel herpå:

Primary result: a complete list of functions



Planning		
Make schedule	Very complex	Update
Calculate schedule consequences	Complex	Signal
Find working hours from previous period	Medium	Read
Enter contents into schedule	Complex	Update
Erase schedule	Simple	Update
…	…	…

Secondary result: specification of complex functions



```
Query possible reservations:
              given time or date or employee-name
              search objects in time period-available and select those
                             who           belong to emplyee-name, is known
                                           have date, if known
                                           cover point in time, if known
              result objects of time period-available that fulfill the criteria
```

- For at finde frem til den komplette funktionsliste anbefales det for uerfarne udviklere at udføre en systematisk analyse med udgangspunkt i de fire funktionstyper. Hver af typerne giver anledning til et antal konkrete spørgsmål, der leder frem til relevante funktioner. Spørgsmålene kan findes på side 140 til 142 i OOA&D bogen.

- Forskellen på events, use cases og funktioner kan være forvirrende ved første øje kast, da de alle beskriver dynamiske handlinger. Forskellen er de forskellige domæner hvori de optræder.
    - Events stiller spørgsmålet: Hvad sker der med objekterne i problemdomænet?
        - ”Ordre placeret” - en kunde placerer en ordre på et specifikt tidspunkt.
    - Use cases: Hvordan vil systemet blive brugt?
        - ”Indtast ordrer” - en bruger i applikationsdomænet skaber en ordrer for en kunde ved brug af systemet.
    - Funktioner: Hvad kommer systemet til at gøre?
        - ”Opret ordrer” - et objekt af ordrer klassen bliver skabt i modellen af systemet.

Der henvises til kapitel 7 i OOA&D bogen for mere dybdegående informationer om ovenstående.

Component Design

Designing the architecture is an iterative process. When we start modeling the components, we might have to come back and revise the architecture. The output of the component design might be something like slide 4.

For the sub-activities of this activity, see slide 5.

We now have problem domain, application domain, and we have specified our general component architecture. Now we are going to specify the architecture.

The model component is the part of the system that implements the problem domain model. Here, we represent events and classes, structures and attributes. We also choose the simplest way we can do this. The result of this sub-activity is a class diagram of the model component seen on slide 10.

Activities: We represent the private events of our different classes. There are two kinds: sequence and selection events marked with a +, and iteration events marked with a *.
Then we represent common (public) events. We are looking for the simplest possible representation, so choose between the alternatives you see.
We then restructure our structure of the problem domain using generalization, association and looking for embedded iteration. This is only really possible if we’ve made changes.

The Bank System

If we look at the event table, we have both private and common events, and sequential and iterative events.

Private events

Sequence and selection events are represented as an attribute in the class described in the state chart diagram. We assign the attribute a value when the event is triggered, and we integrate that attribute in the class (we can do this because we know the event will only occur once).

Iterative events are represented as new classes connected to the class described in the state chart diagram with an aggregation structure. When the event occurs, a new object of that class is instantiated.
Again, integrate the attributes of the event in the class.
If the event doesn't contain any attributes, consider not making a new class. The argument is that the new class doesn't contain any information, so there is no need to create objects to represent the event. This is applicable for "updated" events.

(Question) How do we actually code this?

An iterative event can happen multiple times, and we want to keep track of this. If it was just an attribute, we would lose all previous events. When modeled as a class and objects, we will always have previous information.

See how the private events from the Customer class modeled on slide 15.

Common events

Consider structural connections between objects to model access to attributes.
If the event has a + and a * on the same line (different classes), we represent it with connection to the class, which gives the simplest representation.
If it has the same sign across all classes in a line, consider other representations.

See slide 17. The account opened has a * for Customer, but a + for Account.
The simplest way to do this is to give the Account class an attribute called open-date and close-date. This is the simplest representation.

When the event is iterative for all involved objects, then the event can be represented as the new class(es) under Account on slide 18. We could also choose to model it under Customer, then it would look like slide 19. See that this is more complicated than slide 18.

Restructure classes

The class diagram can often be restructured and simplified without any loss of information. We don't really need two classes Deposit and Withdraw, they can be combined into a Transaction class. Simplify.

Another way is to look for associations. In a gas station, we need to keep track of Customers filling their cars with gas. To model the event that the customer fills the car with gas, this is iterative for both classes, so we need to add another class, which aggregates to both classes. Then we can actually remove the association.

Embedded iterations

See slide 22. We have three events aggregated by Person, which are events modeled as classes. This does not represent the association between the events. You cannot receive treatment after discharge. In cases like this, we choose to move on to hierarchical structure instead. See how the hierarchy makes much more sense as to what is actually happening. The discharge class is modeled as an attribute in the hospitalization class now.

Group assignment

We are doing the group assignment in the lecture. See slide 23 for the class diagram.
The Movie Played event is common and has different signs across classes. This can be represented as an attribute for the Show class, seeing that this is where the event is sequential. The attributes are seen in the attributes for the event itself in the behavioural diagram.
Same thing goes for Show Planned.
Customer closed and opened are both attributes for the Customer class.
Movie Rated can be a new class aggregated by Customer and Movie, that has attributes rate and date. The Movie Viewed event could be a new class, but it’s way more simple to make it an attribute “view date” in the new Rating class. This is given that we only want to keep track of ratings, and not when people watch the movies. If it was the other way around, we could make a new class Viewed with a rating attribute.
Cinema opened and closed are both attributes.

Exam example

For class 1, we have a private sequential event H1 with attribute A1, so the class should have the attribute A1. Furthermore, it should have an attribute called C1State, as we want to keep track of the states of the class, for example when it is destroyed.

For class 2, it should have attribute A4, as well as a state attribute. For H2, it is iterative for both C1 and C2, and as such should be made into a fourth class aggregated by C1 and C2, and should have attributes for the state and for A2.
H6 is iterative for C2 only, and should be made into a new class aggregated by C2. It should have attributes state and A6.
Note that H5 is iterative for C2, and sequential for C3. We give C3 the attribute A5, but as it is iterative for C2, we need to show this connection between the two classes. We do this by letting C2 aggregate C3, which is the class that has the attribute it needs.

For class 3, we want a state attribute. Furthermore, it should have A5, A9, and A10 as attributes.

Always remember multiplicity when doing exam exercises like this.

Exercises