Innlevering 3

Jan Inge Lamo

06.05.16

1 Estimering

a) To estimate the size of system, we simply add together all the use cases for the system. Because the set of use cases is the system requirements. Then give each use case a point based on how difficult the development team perceives them. Based on how many points we end up with, will it give us an estimation on the functional size of the system. Distributing points to use cases can be done in a similar fashion to planning poker, except that user stories is swapped with use cases. Then the management could use a model based on past experience to determine the size of the system from the points. This is of course assuming the team has experience to convert use case points to size.

b)

To estimate the complexity of the system, (assuming the development team has some experience/competence) we can do group opinion with structure, more specifically planning poker. The nice thing about planning poker is that everyone shares their opinion the same time, so people won't be anchored by others. The points for the user stories relate to how difficult it is to implement in the system, and how difficult it is to implement relate to the complexity of the system.

2 Arkitektur

a)

1. *Logical view* show the structural logic between classes/objects in the system. Each view can generally be a use case.

2. *Process view* show the flow between components in the system during runtime. This view is advantageous when checking whether the system fulfills the non-functional requirements.

3. *Development view* show how the different parts of the system is split up into smaller components. This view if for the project manager so they can delegate parts of the system to different development teams.

4. *Physical view* show where and on what the system will be running. In our example with Markasykler, the physical view is the bike stations and the servers that deal with user login and the web page.

b)

It is useful to include different views in describing the architecture because an architect have to consider many aspects when building a system. The views also correspond to what the sub-teams has to focus on. Logical and process views has to do with the software development team, the development view is for the project manager, and physical view concerns the engineers. Having a 4+1 view model is convenient because the views are logical and intuitive.

c)

3 layered logical architecture

User interface on bike station
Markasykler system
Database

4 layered physical architecture

Bike station client
Bike station server
Markasykler server
Database server

I would argue that the bike station is both a client and a server. The client part takes care of user input, while the server part controls the bikes and communicates with the central Markasykler system. We do this to limit the impact a user has on the bike station. Also even though it is not the 80's anymore graphical user interfaces are still prone to crashes compared

to pure data driven systems. When the interface is a client we mitigate potential unwanted behavior from the bike station (like unlocking all the bikes).

d)

Layered architecture is easy[1] to understand even for non-technical people, because a layer is only dependent on the layer directly beneath it. If the interface is maintained, we can swap entire layers, which increases modularity and maintainability.

e)

When designing an architecture it's generally difficult to design layers that depend only on the layer directly beneath it, and if that proves impossible then modularity and simplicity goes out the window. Also a clear distinction which layer a certain component belong to is not always clear. When designing closed layered architecture and a layer need to communicate with a layer that is not directly beneath it, then we use processes on each layer in between

f)

1. $Client-server$ architecture is just a two-layer logical architecture, this architectural pattern is often an abstraction because there are probably more services running on the server for handling client requests and processing the response. When talking about client-server architecture, people mostly mean thin-client, which offloads heavy work to the server.

2. MVC architecture is a pattern with three modules which contain all the software services used in the system. As the name gives away, these modules are; model, view, and controller. View is what the user uses (can be used through a browser), controller handles the input from the view, and the model is the underlying system

3. SOA is the practice of designing small applications, that make use of external services. When the applications only call the services, then it's easy to change services them if the interface is the same. To explain better let's compare SOA to OO-programming, objects are applications and services are methods. Say you have three classes but almost all the methods duplicated on at least one of the other classes.

[1] If done correctly, i.e. closed layered architecture.

The methods are not relatable to the objects so it'd be confusing to make them public, so instead we make a new object(s) with these methods and let the other classes call them from the that instance.

4. *Pipe and filter* architecture is based on the idea that data flows through a system. The common example is the 'pipe' command in UNIX based systems, which allow you to send output from one program to another, linking them together. When this is applied to systems it is data that is sent through different components or services.

3 Aktivitets- og tilstandsdiagram

State diagrams and activity diagrams are both pretty similar. The difference is that state diagrams are finite-state machines, which changes state based on input. While activity diagrams are the flow of a process, what has to be done next on a given action. One way to think about it is that state diagrams is from the computers perspective, while activity diagrams is from the users perspective.

a)

State diagrams help us visualize which state the system changes to with a given input. When examining a state diagram, we can easily determine whether a given input sequence will result in success or failure. State diagrams are used for processes that either accept or reject a given input.

b)

Activity diagrams show the flow in an process. Between each node there is an action that is performed (sometimes just processed). Conditions are handled by diamonds in the model. Activity diagrams can be used to visualize flow in use cases.

c)

See page 6

d)

See page 7

4 Testing

a)

Unit testing focuses on the testing individual classes and methods that are instances of the object. When testing methods we shouldn't only test valid input, but also the unexpected.

Interface testing is done when the units/components/objects are put together. We have to make sure that the interface between object communicate properly.

System testing is controlling that the data is flowing correctly through the system. And that everything works as a whole.

Acceptance testing must be done by the end user, because they're testing if the finished system is acceptable and if it's what they wanted.

b)

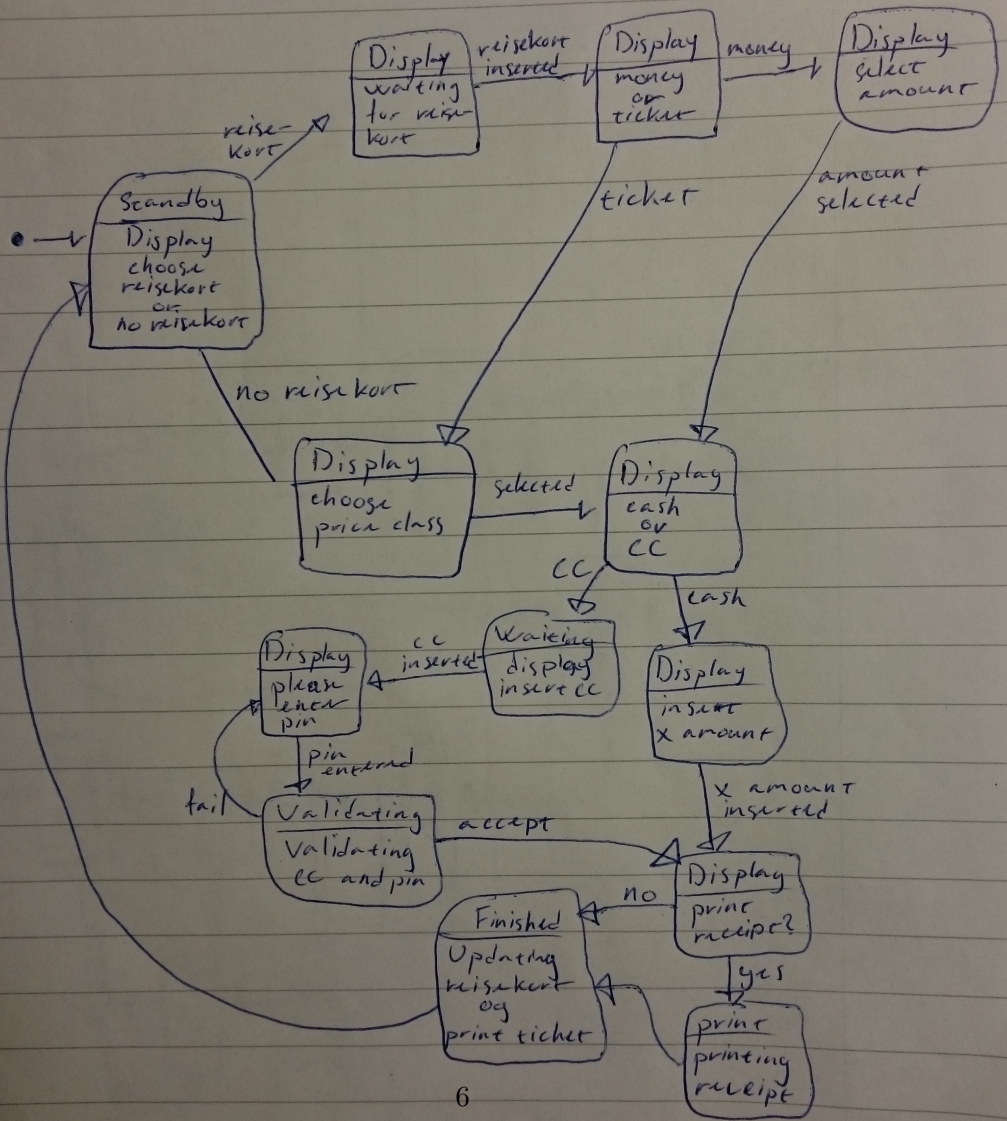
Unit testing is done on the bike station software, on the central Markasykler system, and the database.

Then we do interface testing between our system and Ruter payment system, between bike station interface and bike station functions, between the bike station and the central system, between the Markasykler system and the database.

When we are sure the interface is working, we test the whole system and check if everything works as it should, we also check if all the non-functional requirements are met.

Now we hand it to potential users so they can take both the system and the bikes for a spin.

State machine model for Ruter payment machine



Note true state machines should have one path each. We shouldn't combine the reisekort and ticket paths as shown above.

Activity diagram for Ruter payment machine