This is an excerpt from my group assignment.

SECTION 3: CRITICIZING UML

UML is a highly standardized graphical modelling language focused on the description of large software products. It contains mechanism for the static structure and dynamic behaviour of such complex systems. UML has a notation to reduce ambiguity and confusion throughout a project. UML specifies a standard notation that covers a large part of the design activities that are needed to build systems. It can also be implemented in any programming language and methodology. UML has given every software practitioner a common notation to communicate ideas and move away from the multitude of proprietary notations that we have accumulated over the years.

One major drawback with UML but which is included in every Object Orientated design process is Use Cases. Use Cases helps describe the problems that the user has to solve. UML has Use Case diagrams which are at a very high level and don't give enough detailed description of the overall functionality of the system. Use Cases give a more detailed perspective of the intended behaviour of the system. In this assignment we found it important to develop Use cases especially for the Sequence diagram.

The history of UML is based on the experience of insufficient notations and, most important, a lack of proper standardization. To fulfil the wishes, needs and requirements of the software developer community, UML borrows many ideas from older modelling languages and tries to provide a unified environment.

One of the major problems of UML is that it is almost impossible to merge different approaches and attempt to create a new language able to do everything: you have to give up specialised features of the original notation and therefore lose some of its power. On the other hand, each UML diagram increases the number of symbols used and raises the effort needed to learn the language.

Therefore the very first model of a system is often created in a stripped down modelling language: a couple of circles and lines between them. UML often comes only as the second step, not the first one. The problem can be that you go directly from a problem to classes - methods and data. There is leap into too much structural detail, because of the level of detail there is conflicting ideas on how to go ahead and develop the models. Its is also hard to see where the responsibilities for each class lie.

Even though there is some software on the market, we had no possibility to automatically translate the UML description into working code. This is essential for the success of a modelling language that is such closely coupled to the actual code. A roundtrip feature (forward and reverse engineering) is essential, too. In our case to make sure the Class diagrams are complete the code was written in C# to ensure that all the attributes, operations and relationships are correct and accounted for.

Another issue is to try and ensure the consistency between the UML diagrams. UML consists of nine types of diagrams. The range of diagrams can leave the overall design specification in an inconsistent state. By using a CASE tool like IBM's Rational Rose this can be managed as changes in one type of diagram can be reflected in other diagrams.

There is no way to guarantee that a UML model fully complies with the requirements stated by a customer. A human could examine the model, though, but we think this process has to be automated. Large projects are decomposed into smaller, more manageable pieces called subsystems. The interfaces between subsystems are important to the integration into the larger whole that is the system. Sequence diagrams are used to specify the interactions between classes on the borders of these subsystems

One of the team members, Stephan Brumme, has some experience with the Fundamental Modelling Language (http://fmc.hpi.uni-potsdam.de) supported by the market leader in business software: SAP. This language explicitly reduces the numbers of distinct diagrams and the number of distinct shapes, too, by strictly using bipartite graphs. It emphasizes the dynamic aspect of software and clearly allows extending and changing a model. The reality shows that requirements often change after their initial settings and have to be adjusted. This is quite hard with UML because most UML diagrams would have to be completely redrawn.

FMC also takes into consideration that a modelling language is meant for humans. In conclusion, a modelling language should not be too close to actual code. Instead, it should reside on an abstract level above the code and is allows hiding several insignificant aspects. The hierarchical refinement of UML diagrams is insufficient, even package diagrams are not abstract enough.

There are some experiments done by visualisation researchers on the usability of graphical notations. The well-known visual variables by Jacques Bertin (Semiology of graphics: diagrams, networks, maps. 1967) describe basic attributes used to create diagrams in general. Scientists in the field of geography heavily rely on diagrams and maps and, hence, have several hundred years of experience with graphical languages.

One of the main facts is that humans have a limited memory and a limited ability to distinguish entities. To transport a huge amount of information, as required by software modelling languages, these facts have to be taken into consideration. For instance, most people cannot perceive more than seven distinct objects. Most UML class diagrams contain far more than only seven classes. To circumvent this problem, a proper use of colours, shapes, size and many more may help. Unfortunately, there is only a restricted use in UML: slight variations of fonts (bold, italics, etc.) are just not enough to aid the construction of a mental map of UML diagrams. Replacing straight lines by slightly curved lines and substituting sharp corners by round corners are subtle ways to make a distinction between shapes. It turned out that using colours can drastically increase the meaning of a diagram. This does not change the contents at all – it merely strengthens what UML is meant for: to make software systems for understandable for humans.

Moreover, UML diagrams do not make consistent use of certain graphical notation: arrows can stand for inheritance (class diagrams) or temporal order (state diagrams).

In the next version of UML, software engineers should sit together with experts in the field of psychological and visual research in order to optimize the notation.

UML forces you to talk in their language not in the business language of users. The analyst must translate what the user wants into another language which may lead to the misinterpretation of requirements when translated into UML notation.

Everyone has their own interpretation of UML and the process to create the UML work products. Therefore trying to agree on the static representation of the application is a non-trivial task. Then in addition the interaction of object with each other is another layer of complexity added to the assignment. We noticed that I our earlier sessions that one person had to drive the process in order to make progress.

Models are not fully operating software systems and shouldn't be treated as such. They are a positive start and by no means a guarantee of success. For example UML cannot control or tight coupling and typically bad design like public attributes in classes for example public static non-final variables. Using Design patterns within UML are an extremely useful mechanism to document and learn about common reusable design approaches. Using design patterns you can reduce the designing time for building software systems and more importantly ensure that your system is consistent and stable in terms of architecture and design. The UML class diagrams provide an easy way to capture and document Design patterns. UML doesn't enforce reusable, loosely coupled design so it is left up to the skill and creativity of the people involved in the project to achieve those goals.

UML notation can be unwieldy at times, especially stereo-typing and other textually expressed relationships, but it is a step in the right direction. Any tool for system design and _expression must be sufficiently extensible as to not slow the blinding pace of development while being standard enough to be universally understood. UML goes a long way towards accomplishing this, and there will undoubtedly be many iterations of its design. However, ultimately any notation will have its tradeoffs, just like any system.

The language surely offers a lot of great features and seems to be the first widely accepted standard. It can never be said enough how important standards are: they allow software developers from different countries, with different background and different experience to work together and share a common view on a model. We think that UML is a good modelling language but there is still room left for further improvements.