Software Engineering Project Workshop (SENG202)

Matthias Galster

Phase 1 – tasks (part 3)

July 16, 2020

UNIVERSITY OF CANTERBURY
Te Whare Wānanga o Waitaha
CHRISTCHURCH NEW ZEALAND

Deliverables

- Project setup checklist
- Design document
- Reflections and logs
- Presentation

Reminder

Executive summary

1. Business and system context
2. Stakeholders and requirements
3. Acceptance tests
4. GUI prototypes
5. Deployment model
6. Detailed UML class diagram
7. Risk assessment
8. Project plan

References

Appendix

Documents are "linear", how you work is not

Content of design document

Deployment model

- High-level UML deployment diagram

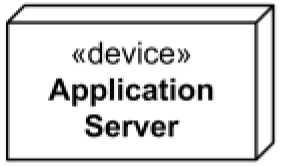
- Physical deployment of software artifacts on nodes
  - Nodes (physical hardware to deploy system)
    - Monitor
    - Modem
    - Servers (e.g., caching, application, DB)
    - Clients
    - Etc.
  - Relationships
    - Physical link between nodes (e.g., TCP/IP, Ethernet, JDBC, REST, RMI)
  - Software artifacts
    - Execs, binary, archive, DB, source, libs, packages, files, documents, scrips, etc.

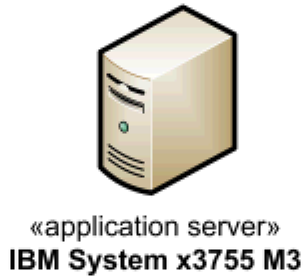
Provide insights

- About the system
  - What hardware resources will you need
  - What is the relationship between resources, how do they communicate
  - What are software resources, where are they deployed
  - Where does your system run
  - What does the user of your app need
- About quality attributes
  - Performance
  - Scalability
  - Maintainability
  - Portability

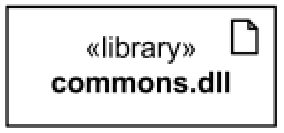
Elements of deployment diagrams

- Nodes
  - Described as boxes
  - Can have sub-nodes (nested boxes)
  - Single node could represent multiple physical nodes (e.g., server cluster)



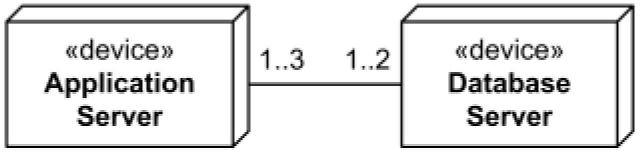


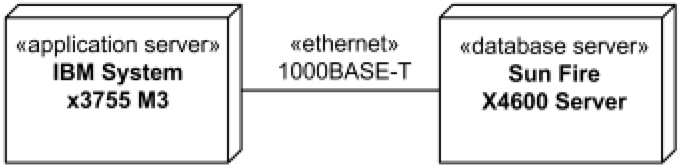
- Artifacts allocated to node
  - Rectangles within boxes

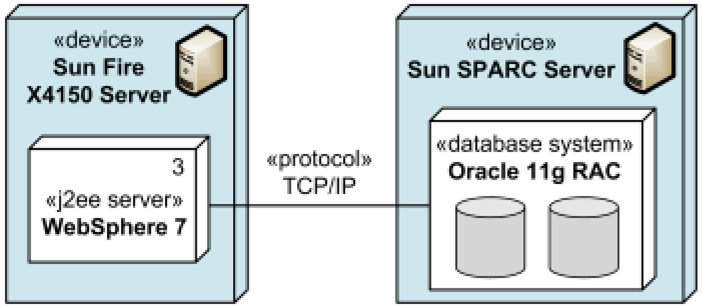


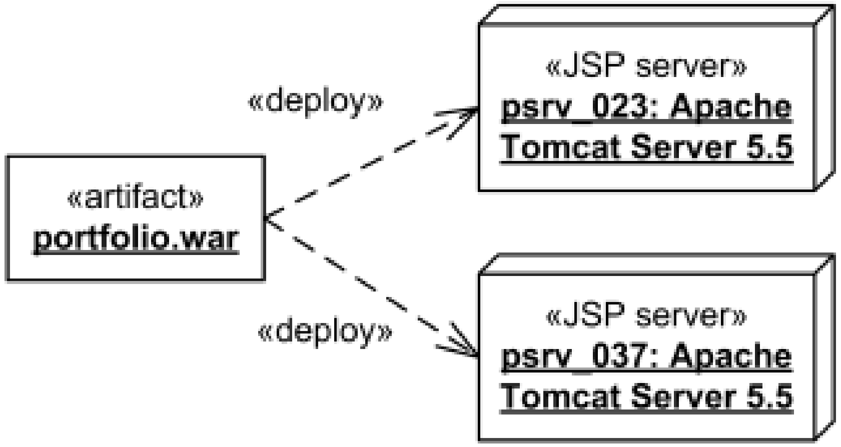
Notation may vary between tools

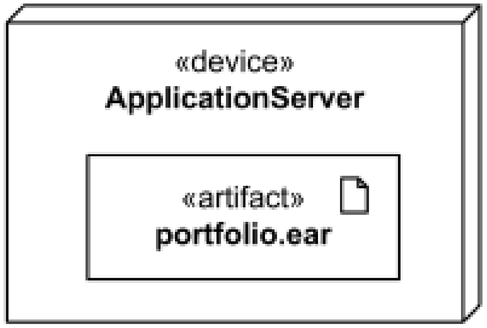
Deployment and relationships



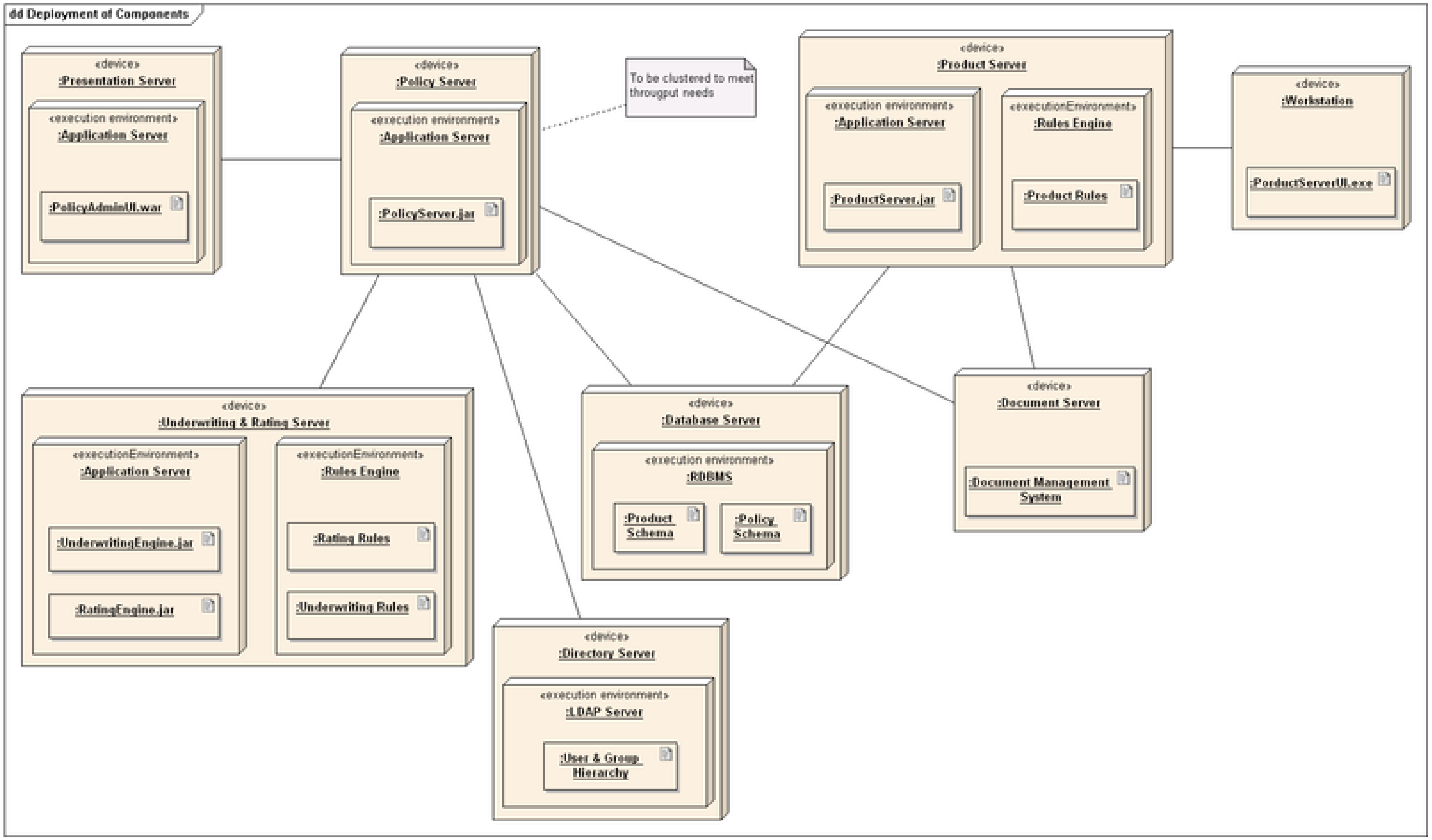





«artifact»
portfolio.war
«deploy»
«deploy»
«JSP server»
psrv_023: Apache Tomcat Server 5.5
«JSP server»
psrv_037: Apache Tomcat Server 5.5



Example (without relationships)


dd Deployment of Components
«device» :Presentation Server
«execution environment» :Application Server
:PolicyAdminUI.war
«device» :Policy Server
«execution environment» :Application Server
:PolicyServer.jar
To be clustered to meet througput needs
«device» :Product Server
«execution environment» :Application Server
:ProductServer.jar
«executionEnvironment» :Rules Engine
:Product Rules
«device» :Workstation
:PorductServerUI.exe
«device» :Underwriting & Rating Server
«executionEnvironment» :Application Server
:UnderwritingEngine.jar
:RatingEngine.jar
«executionEnvironment» :Rules Engine
:Rating Rules
:Underwriting Rules
«device» :Database Server
«execution environment» :RDBMS
:Product Schema
:Policy Schema
«device» :Document Server
:Document Management System
«device» :Directory Server
«execution environment» :LDAP Server
:User & Group Hierarchy

Hint

Deployment models describe how and where data is stored, how the system and its runtime components are distributed across resources, etc. The deployment model should include a textual description of what processes will run on what computers and how they talk to each other. Nodes are typically physical hardware. In the case of a desktop app, there is probably a local PC. The next question is what executables are required, other libraries, files, etc. For example, there is probably an operating system, a JRE and the executable of the app. These are probably deployed on the local PC. There app may require some libraries (or API's, external jar's), e.g., for Google Maps. Where do they live? There are probably also files (xml, text, etc.) to store data, app-specific data or some other resource information. Where do these files live? There might be a database or a database server? Where is this database / server deployed? Are there other software resources? If yes, where to they live? You may also think about the features that involve a server with an "external" database. This could mean that you host another database / database server on a separate physical machine. If you do that you probably have some sort of software "client" (i.e., another executable) that is running on your local PC (with your app) so that you can connect to that other physical machine. This also means that this other physical machine needs to be able to talk to your "client", i.e., you need some sort of "server" software on the other machine. This "server" software could then communicate with the actual database / database server on that separate machine.

UML deployment diagrams sometimes describe the same thing in different ways so we won't be too picky about the notation and syntax as long as you use one possible UML deployment model notation.

See also www.uml-diagrams.org/deployment-diagrams.html

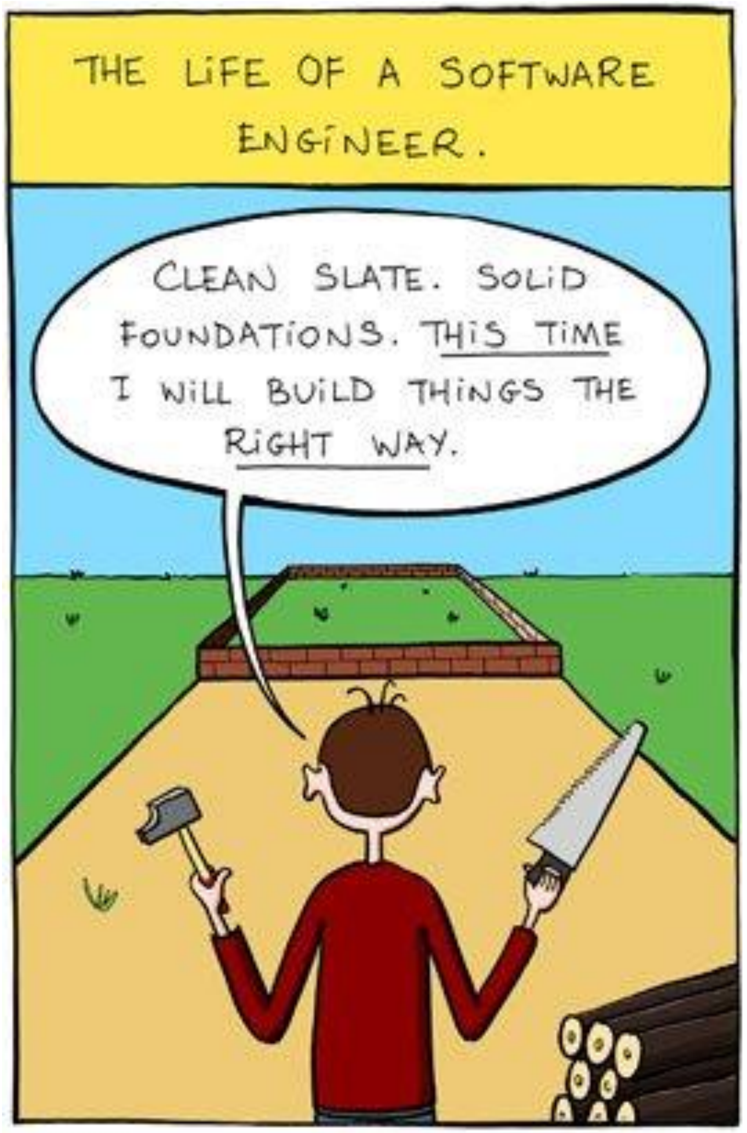
Content of design document

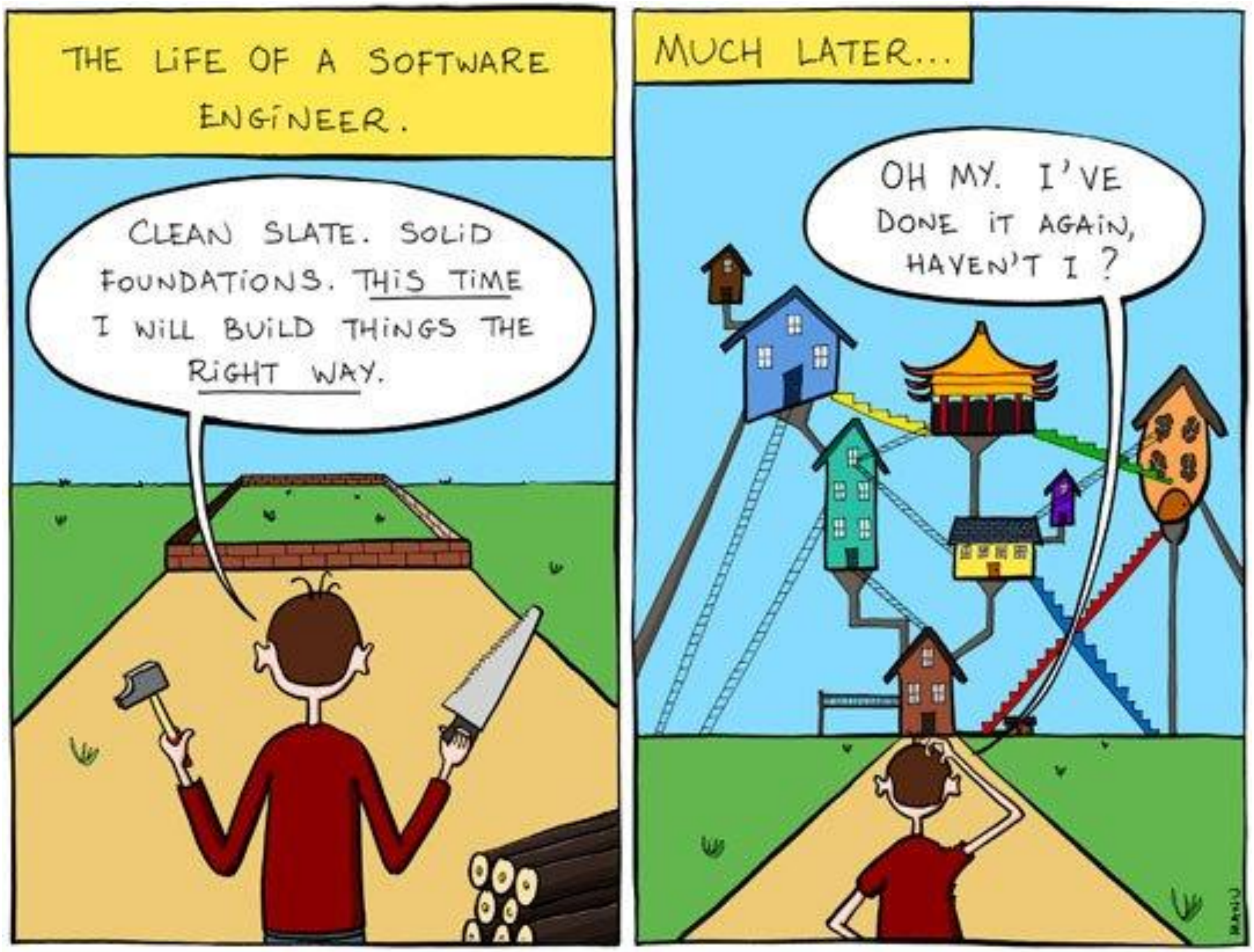
Executive summary

1. Business and system context
2. Stakeholders and requirements
3. Acceptance tests
4. GUI prototypes
5. Deployment model
6. Detailed UML class diagram
7. Risk assessment
8. Project plan

References

Appendix


THE LIFE OF A SOFTWARE ENGINEER.
CLEAN SLATE. SOLID FOUNDATIONS. THIS TIME I WILL BUILD THINGS THE RIGHT WAY.


THE LIFE OF A SOFTWARE ENGINEER.
CLEAN SLATE. SOLID FOUNDATIONS. THIS TIME I WILL BUILD THINGS THE RIGHT WAY.
MUCH LATER...
OH MY. I'VE DONE IT AGAIN, HAVEN'T I ?

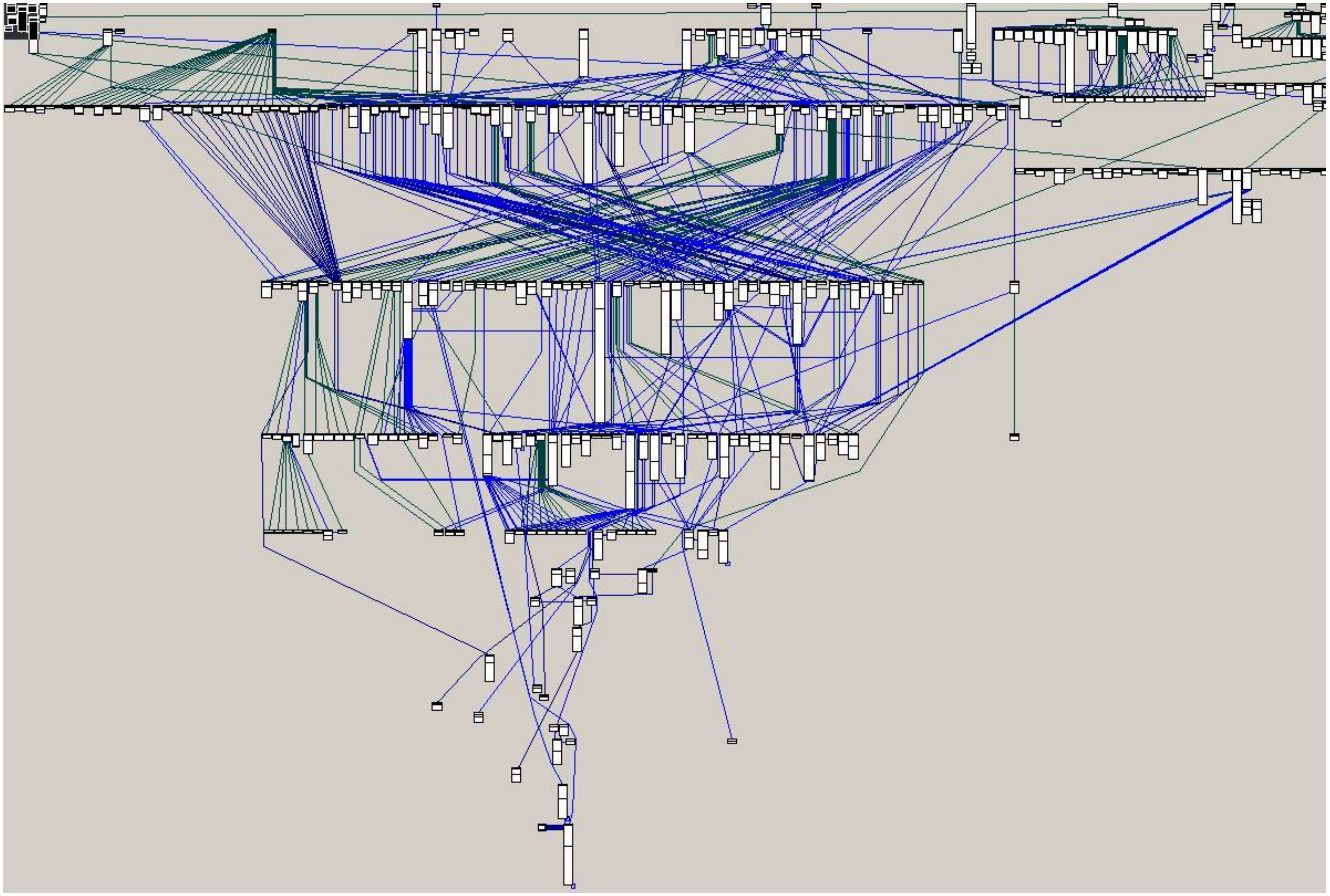
Class diagrams

- Include all major classes and relationships
  - Should help you design, not be produced after design finished
- Use design patterns and principles
  - Information hiding, MVC, Observer, etc.
  - Will require packages, interfaces, etc.

- Will be refined
- Will change

- Keep diagrams from becoming too big
  - Decompose if needed, watch layout and graph readability
- Typical issues
  - Completeness? GUI details? Pitfalls of object-oriented design

Class diagram aesthetics



Hint

Create a UML class diagram, including all major classes and their dependencies and relationships among classes (e.g., aggregation, inheritance). In contrast to the domain model, the UML class diagram also includes non-domain-specific aspects, such as communication (client classes, server classes, data container classes, etc.). You may want to use design patterns (e.g., MVC) to design your architecture. In this case, your class diagram needs to include at least all model and controller classes. It may be reasonable to create class diagrams for different components of your system (e.g., model, view and controller classes that are placed in different packages). In case you decide to create more than one class diagram, please document the component structure. The diagram(s) should be reasonably complete in that associations etc. alluded to in the class instances or use-cases should be shown in the diagram. However, also try to keep the diagrams from becoming so big and the edges so tangled that it is not readable. An unreadable diagram is a useless diagram. For instance, showing lots of methods in the classes, and thus making them really big, is generally not a good idea. Also, if things are getting messy, diagram each package separately so there are many small diagrams instead of one large diagram. Include package information (added to class diagram to show what classes are in what package).

Hint

Completeness: This is a challenge because there is no clear “goalpost”. The target is to have classes proposed for the core of the design, with as many methods as you can think of put on them. You may want to look at your GUI sketches, use cases, and class design to make sure there are use cases for all GUI actions and classes (including appropriate methods) assigned the responsibility of the use case actions.

Do not include all the details for GUI classes (e.g., buttons, listeners). You can include main GUI elements as classes only (e.g., the app will probably have a “main view”) without properties and methods. You may have additional GUI classes that are not shown in the class diagram. You can mention this in a brief textual description that complements the class diagram. If you would like to add more details about the GUI classes to the class diagram, you can add a separate class diagram or put them in a package.

Avoid pitfalls of poor object-oriented design. Such pitfalls are also often referred to as “design smells” or “code smells”. In contrast to bugs, smells are not technically incorrect and do not currently prevent the program from functioning. Instead, they indicate weaknesses in design that may be slowing down development or increasing the risk of bugs or failures in the future. Examples include data-centric designs (e.g., by focusing on “data classes” with setters and getters only), large classes that result in “god objects” (i.e., objects that know or do too much), lazy classes (i.e., classes that do too little), feature envy (i.e., a class that excessively uses methods of another class), high coupling and low cohesion.