NASA Employee Repository Design Document Notes

Lee Saxton - Blue
Celine Teh - Purple
Kwan Lee - Orange

Tutor's Skype: cathryn.peoples
Domain

- NASA employee repo
- Local database, who can access (on-ground control)
- Meeting Minutes 3, 4 (pre), 5 (with tutor), 6 (post)

System requirements

- OS – Linux Ubuntu-Kernel (Morales, 2022)
    - README – setup instructions and how to execute
- Windows OS
    - README – setup instructions and how to execute
- Private APIs for interdepartmental sharing (Brooks, 2013)
- Database - MariaDB? (open-source updated counterpart to MySQL) – NO (Ankush, 2022)
    - Do we want a UI for this project? No, not for this project.
    - **SQL** is the winner!
- Logging libraries (Solarwinds, 2022)
    - https://coralogix.com/blog/python-logging-best-practices-tips/

Assumptions

- Data download requirements – 8GB RAM, 2MB per min
- CPU – quadcores
- API - SOAP
- Latest Windows/Linux OS distribution (compatibility)
- Monolithic prototype
- Above are mostly resources, need to hash out assumptions
- Context/story/situation, who are using the system, what data being accessed, security measures/concerns
    - E.g. Astronauts on the moon accessing rover data from a local database on the ground staff
    - Monitor vitals, temperature etc

  - Don’t focus too much on functional requirements, prioritise non-functional requirements (specifically security)
    - Demo the high priorities (security)

Limitations

- Request/reply is limited to local ping
- Open-source programs only
- Limited CPU/storage for prototype
- Command line based – No GUI/UI

Security Frameworks

- Error Messages
  - Customized for minimal info disclosure
    - 200
    - 300
    - 400
    - 500

- Access controls (Negus, 2020)
  - Interdepartmental Groups
    - NASA employee - no privs
      - SpaceStation - no privs
        - SSAdmin - read
      - HR/AdminDept - read
        - HREmployees - read, write
      - IT - read? Or just execute?
        - ITAdmin - execute
    - Employee
    - NASA Employee
      - Astronaut/Pilot
      - Researcher/Engineer
      - Medic
      - Accountant/Marketing
      - HR/IT/Admin

  - Group NASA – read
    - Admin – read, write, execute
    - NASA HR – read, write
    - NASA Admin OTHER – read
    - Other – no privileges
  - Group OtherAgency – read

        - All other relevant government department admins would be added
    - Other - No privileges

Local scope

- Session Management (Pinto & Stuttard, 2013)
    - Cookies – with encryption
    - Inactivity – automatic logout
- Authentication Management (Pinto & Stuttard, 2013)
    - 2FA – saved credentials
    - Login timeout – automatic reset with saved credentials (only if we have time)
- Input restrictions (Pinto & Stuttard, 2013)
    - Only allow certain charsets for certain input fields
- Boundary validation (Pinto & Stuttard, 2013)
    - Validation checks/input sanitation at every processing layer looking for specified attacks at each layer
    - Server-side
    - Multi-step validation and canonicalization?
        - Can avoid tricks like <scr<script>ipt>

Tools

- Source code parsing (Li, 2021)
    - Python code parsing for weaknesses and errors
- Code testing
    - Unit testing - each function/method/feature
    - Integration/end-to-end testing

Libraries

- Python programming language
- pylint, pytest, pandas, logging, sqlite3
- Windows, Linux
- Input sanitisation (user input field validation): SQL, XML, HTML, encoded characters
- Source code parsing
- Security threats: SQL injection, XSS, XML, unauthorised privileged access (IAM boundary), buffer overflow, cookie tampering, info leak via logs
- MoSCoW
    - Must have: login
    - Should have: check input
    - C and W: in appendix
- Add: Encryption, data storage, regex
- Prioritise discussions on security

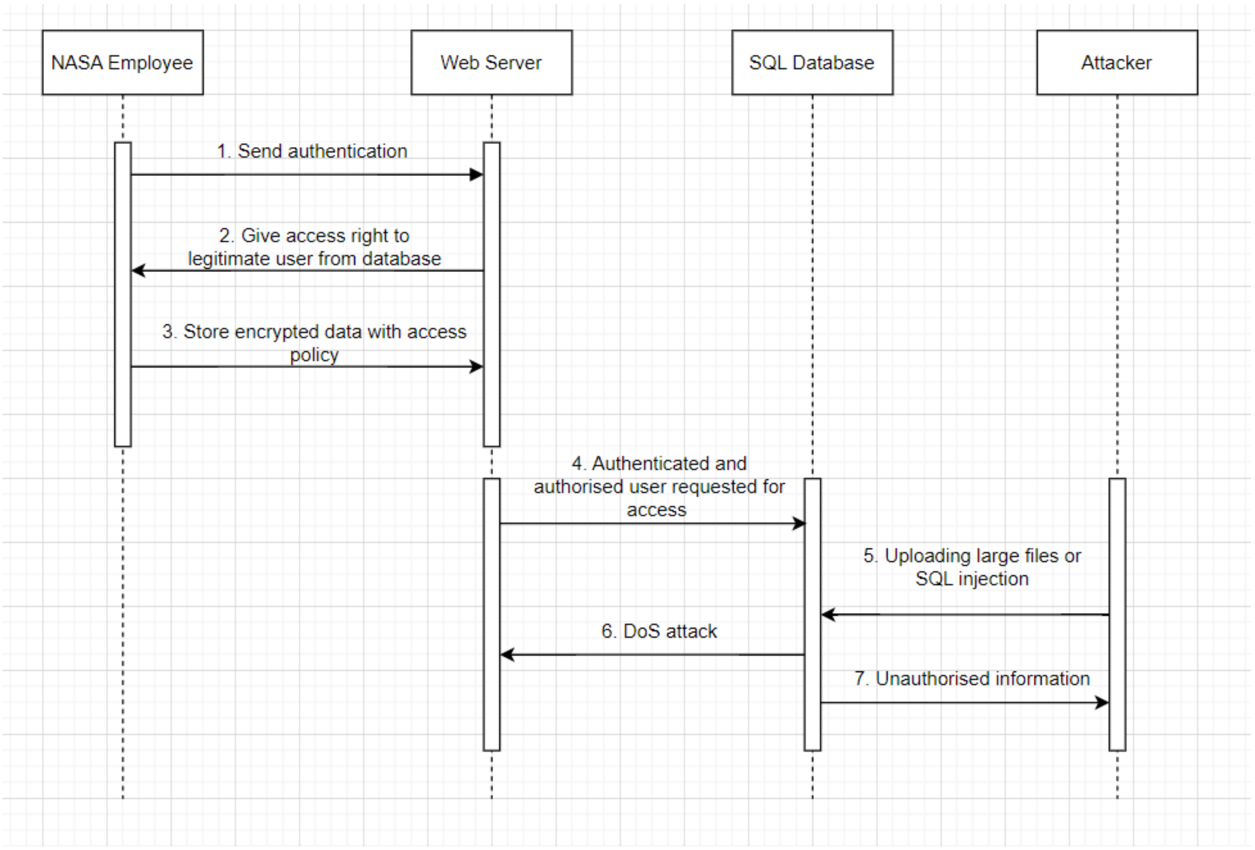
- Object-oriented
- DB: row-based access control
- STRIDE, OWASP to structure vulns and mitigations

Jamboard

Brainstorm Visualisation (SQL and access control brainstorm)

UML Diagrams

- **Create Sequence Diagram (database attack, e.g. injection)**


NASA Employee
Web Server
SQL Database
Attacker
1. Send authentication
2. Give access right to legitimate user from database
3. Store encrypted data with access policy
4. Authenticated and authorised user requested for access
5. Uploading large files or SQL injection
6. DoS attack
7. Unauthorised information

- **CISCO**

A database attack sequence typically includes the following steps:

- **Reconnaissance: The attacker gathers information about the database, such as the type of database, the operating system it is running on, and the network architecture. This information can be obtained through port scanning, network sniffing, social engineering and other methods.**
- **Vulnerability scanning: Attackers scan databases for vulnerabilities such as weak passwords, unpatched software, and misconfigured settings. This can be done using automated tools that scan databases for known vulnerabilities.**

- Exploitation: Once a vulnerability is discovered, the attacker uses various techniques to exploit the vulnerability and gain access to the database. This could include SQL injection attacks, buffer overflow attacks, or privilege escalation attacks.
- Privilege Escalation: Once access to the database is gained, the attacker tries to gain higher privileges, such as administrative privileges, to control the database and access sensitive data. login information.
- Data breaches: Once attackers gain access to databases and sensitive data, they may attempt to exfiltrate the data for later use or sale. This may involve copying the data to a remote server or using other techniques such as steganography to hide the data in other files.
- Cover tracks: To avoid detection, attackers may attempt to cover their tracks by deleting logs, modifying system files, or using other techniques to hide their presence on the system.

OWASP Framework

The OWASP framework is based on the principle that security should be built into web applications from the ground up, rather than being added as an afterthought. It provides a comprehensive set of guidelines and best practices for each stage of the software development life cycle, from requirements gathering to deployment.

The OWASP framework includes a list of the top 10 web application security risks, which covers the most common vulnerabilities that can be exploited by attackers. These risks include injection flaws, broken authentication and session management, cross-site scripting (XSS), and insecure cryptographic storage. By addressing these risks, developers can greatly improve the security of their web applications.

The OWASP framework also includes a number of tools and resources that can be used to test and assess the security of web applications. These tools include the OWASP ZAP (Zed Attack Proxy), which is a popular open-source web application security scanner, and the OWASP Top Ten Proactive Controls, which provide guidance on how to proactively address the top 10 web application security risks.

- potential mitigation + background info

SQL injection attacks are a type of cyber-attack that target databases by exploiting vulnerabilities in web applications that interact with them. The goal of these attacks is to manipulate the SQL statements that are used to communicate with the database, allowing attackers to gain access to sensitive data, modify data, or even take control of the database server.

SQL injection attacks work by inserting malicious SQL code into a web application's input fields, such as search boxes or login forms. This code can be used to modify the SQL statements that are sent to the database, allowing attackers to bypass authentication mechanisms, steal data, or execute unauthorised commands.

For example, an attacker could use SQL injection to bypass a login form and gain access to an application's administrative console. They could then modify data or insert new data into the database, potentially causing damage or stealing sensitive information.

SQL injection attacks can be prevented by implementing strong security measures, such as input validation and parameterized queries. Input validation involves verifying that the data entered into input fields is of the correct type and format, and does not contain malicious code. Parameterized queries involve separating the SQL code from the user input, ensuring that input data is not interpreted as SQL code.

Regular security testing and vulnerability scanning can also help identify and prevent SQL injection attacks. By being aware of the risks and taking appropriate security measures, organisations can protect their databases and prevent potentially devastating data breaches.

o

§ **List of attacks on database**

1. SQL injection attack: This type of attack is to inject malicious SQL code into the input field of the web application to manipulate the database and steal sensitive data

2. Denial of service (DoS) attack: This type of attack works by flooding the database server with traffic in order to overwhelm its resources and interfere with its operation.

3. Password attack: This attack method uses brute force or dictionary attack to crack weak passwords and gain access to the database.

4. Malware attack: This attack consists of installing malware on the database server to steal data, modify data, or disrupt database operations.

A potential design for a NASA employee repository might include the following components:

User Interface: The repository needed an intuitive and user-friendly interface that would allow authorised users to access and manage employee information. The user interface will be designed to be responsive and mobile-friendly, allowing users to access the repository from any device.

Authentication and authorization: The repository will be secured with user authentication and authorization mechanisms to ensure that only authorised users can access and manage employee information. This will include multi-factor authentication, password policies and user rights management.

Employee Profile: Each employee will have a profile including their personal information, title, department, contact information and other relevant details. These profiles can be searched and sorted by name, department, title, and other criteria.

Document management: The repository will allow users to upload and manage employee-related documents such as resumes, performance reviews, and disciplinary records. These documents can be searched and sorted by type, date and other criteria.

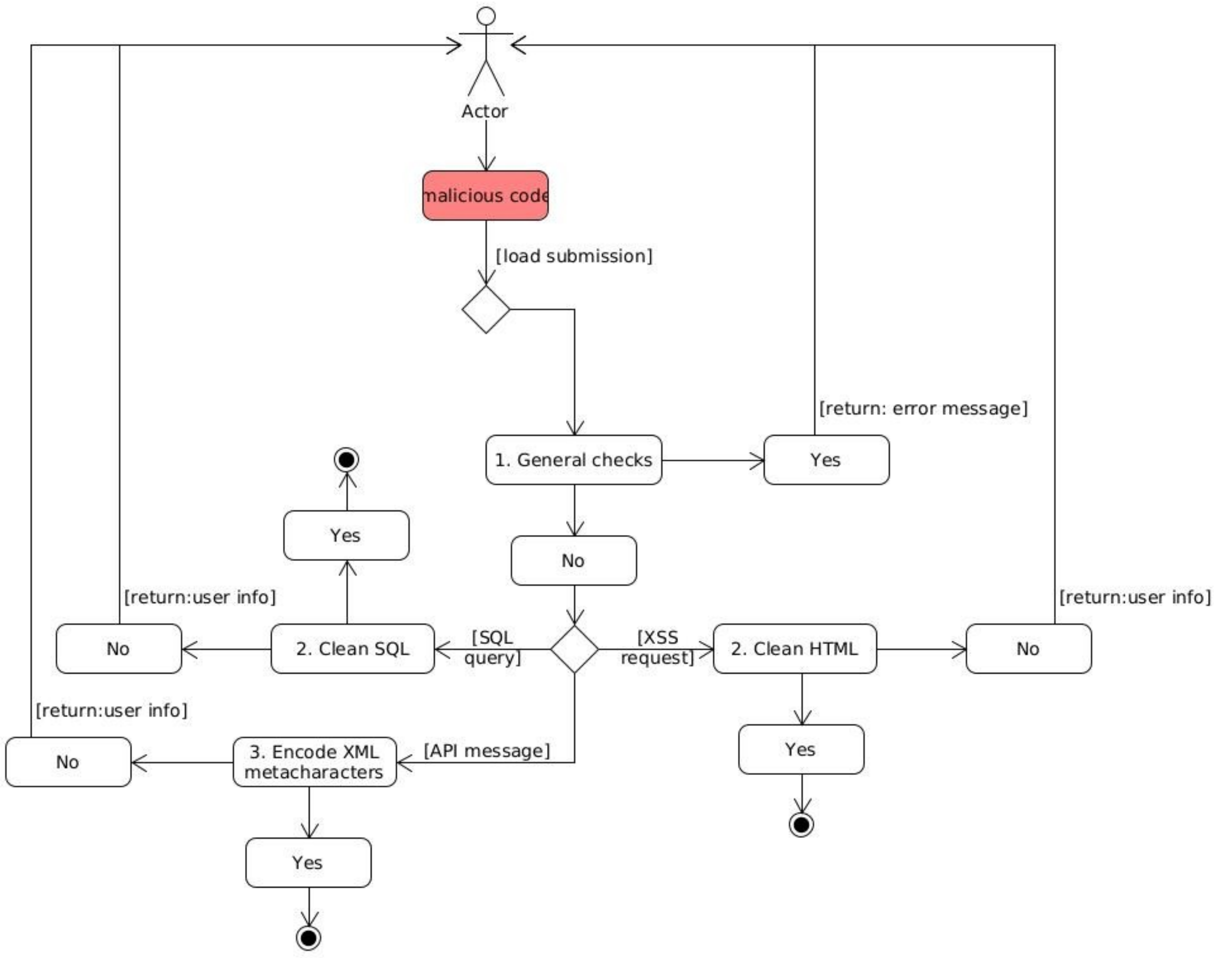
Reporting and analytics: The repository will include reporting and analytics capabilities that allow managers and administrators to generate reports on employee data such as headcount, turnover rates, and performance metrics. These reports can be customized and exported in a variety of formats.

Integration with HR systems: The repository will be integrated with other HR systems, such as payroll and benefits management software, to ensure employee data is accurate and up-to-date. This will require a robust API and data synchronisation mechanism.

Data Privacy and Security: The repository will comply with all relevant data privacy and security regulations, such as GDPR and HIPAA, and will include data backup and disaster recovery mechanisms to ensure data integrity and availability.

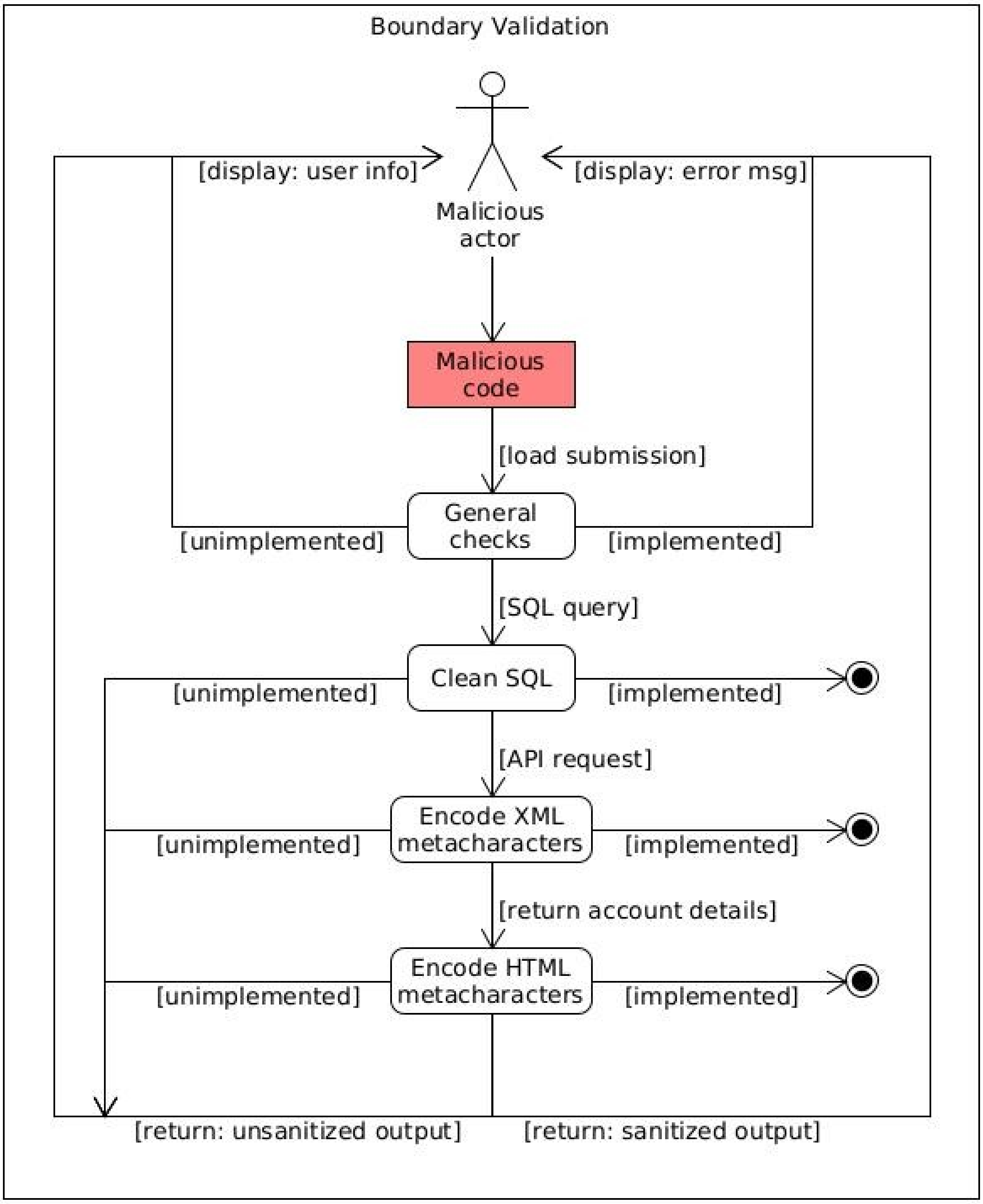
Overall, the design needed to balance usability, security, and compliance requirements to ensure NASA employees and administrators could easily access and manage employee information while protecting sensitive data.

Activity diagram (first draft – still need to add the boundary layer and get rid of the yes/no’s)



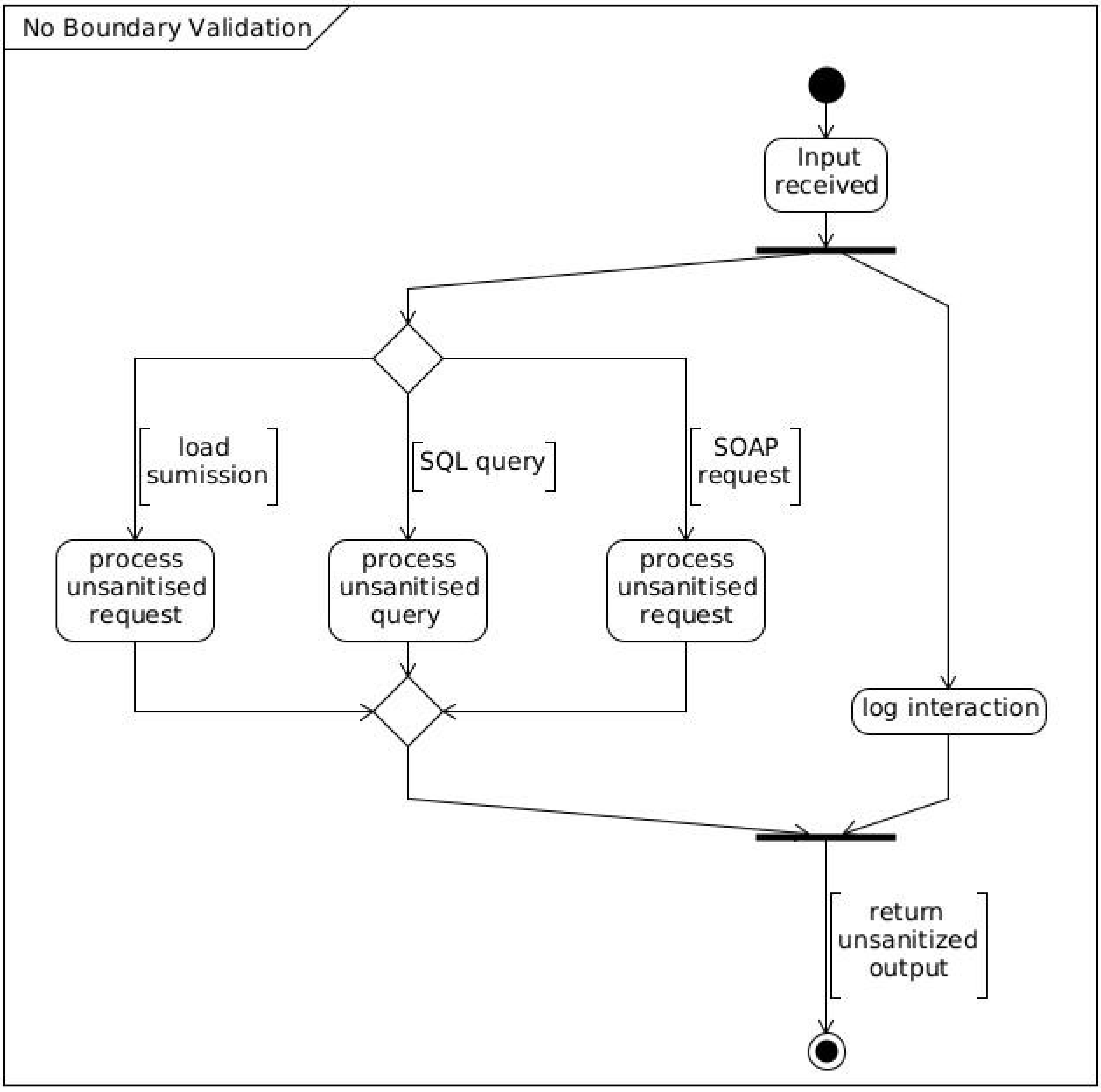
Activity diagram (second draft)

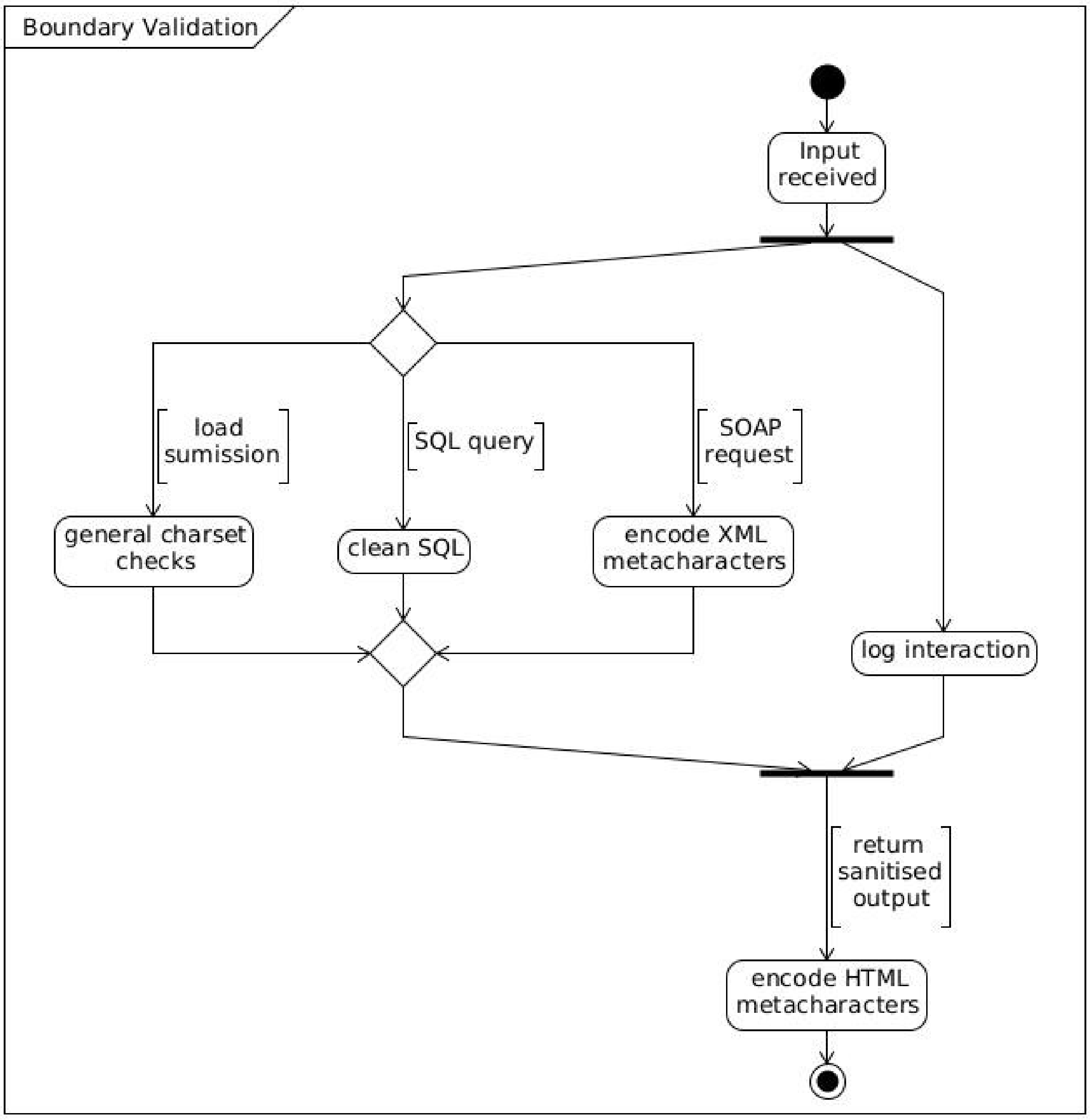
Activity Diagram (Third draft)



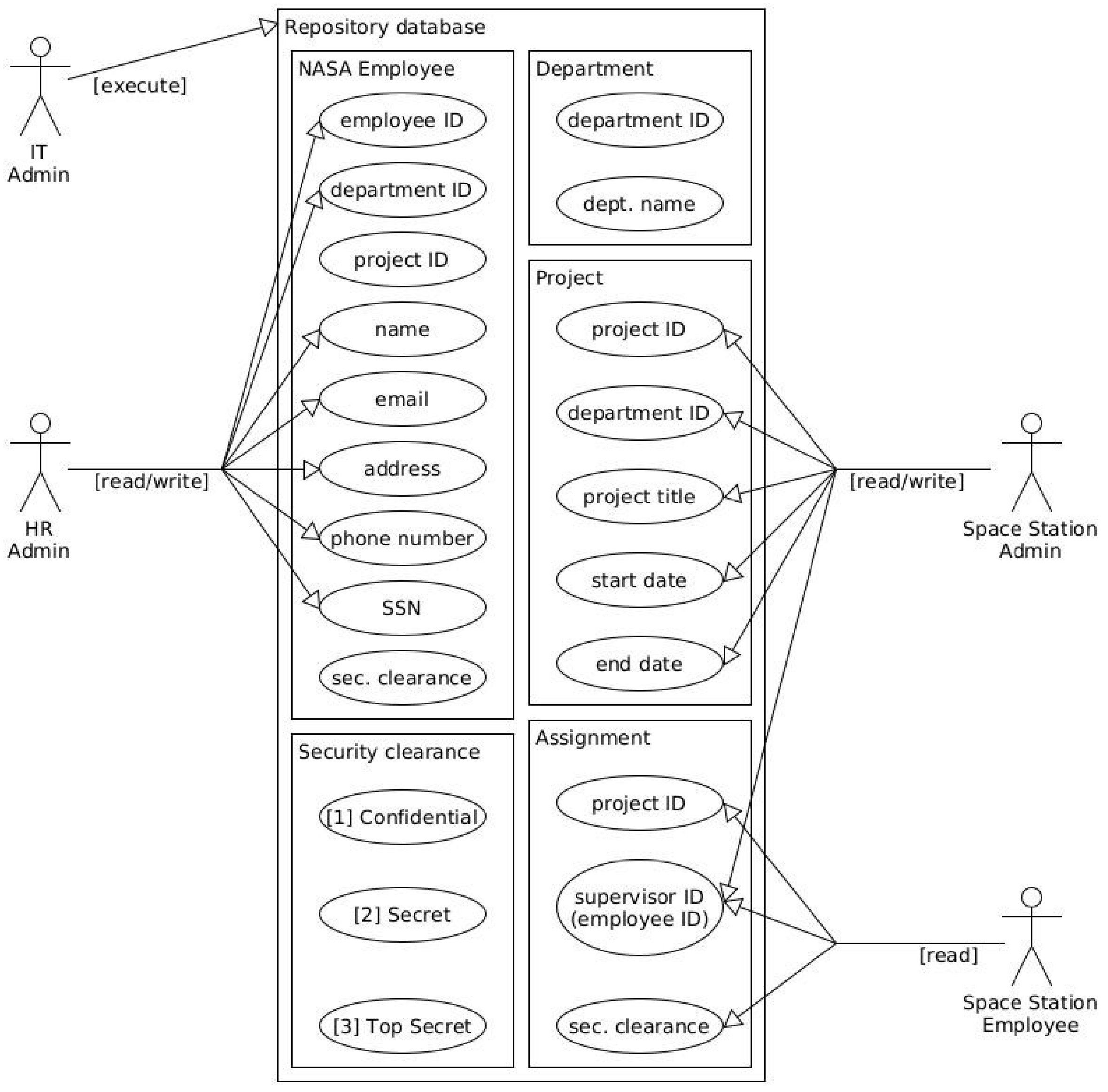
Feedback: This is more of a process/flow diagram (include reference), note that this is not UML-recognised. Activity diagram is somewhat the equivalent in UML. System startup, data entry, clean code.

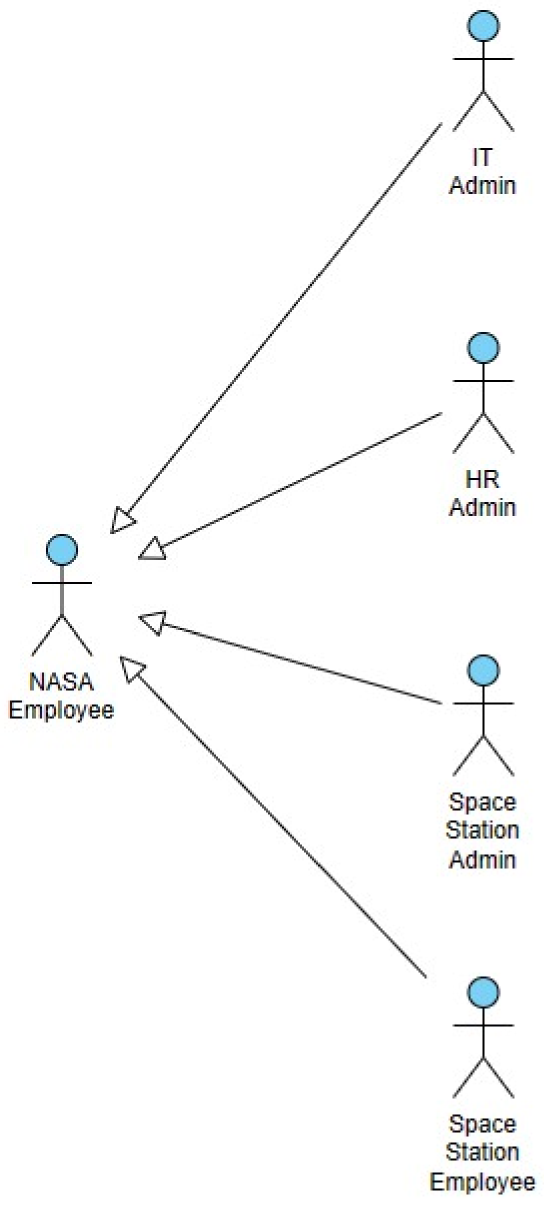
Activity Diagram (Third draft)


No Boundary Validation
Input received
load sumission
SQL query
SOAP request
process unsanitised request
process unsanitised query
process unsanitised request
log interaction
return unsanitized output

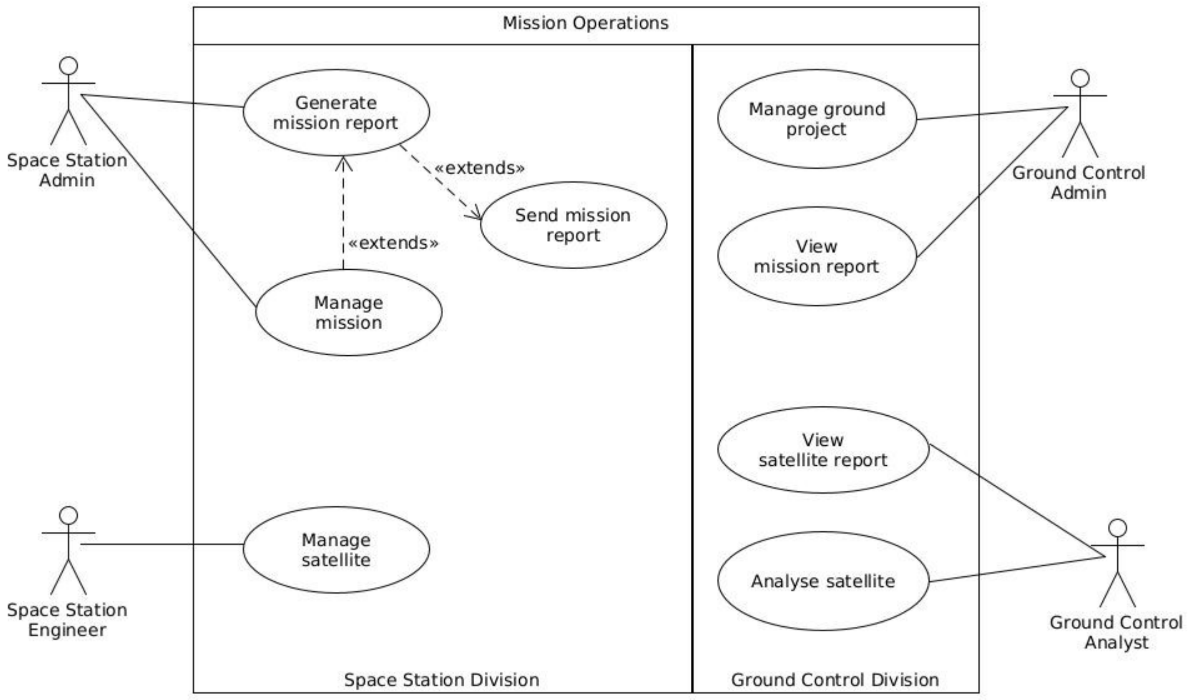


Repository Use Case (first draft)


Repository database
NASA Employee
employee ID
department ID
project ID
name
email
address
phone number
SSN
sec. clearance
Department
department ID
dept. name
Project
project ID
department ID
project title
start date
end date
Security clearance
[1] Confidential
[2] Secret
[3] Top Secret
Assignment
project ID
supervisor ID (employee ID)
sec. clearance
IT Admin
[execute]
HR Admin
[read/write]
Space Station Admin
[read/write]
Space Station Employee
[read]

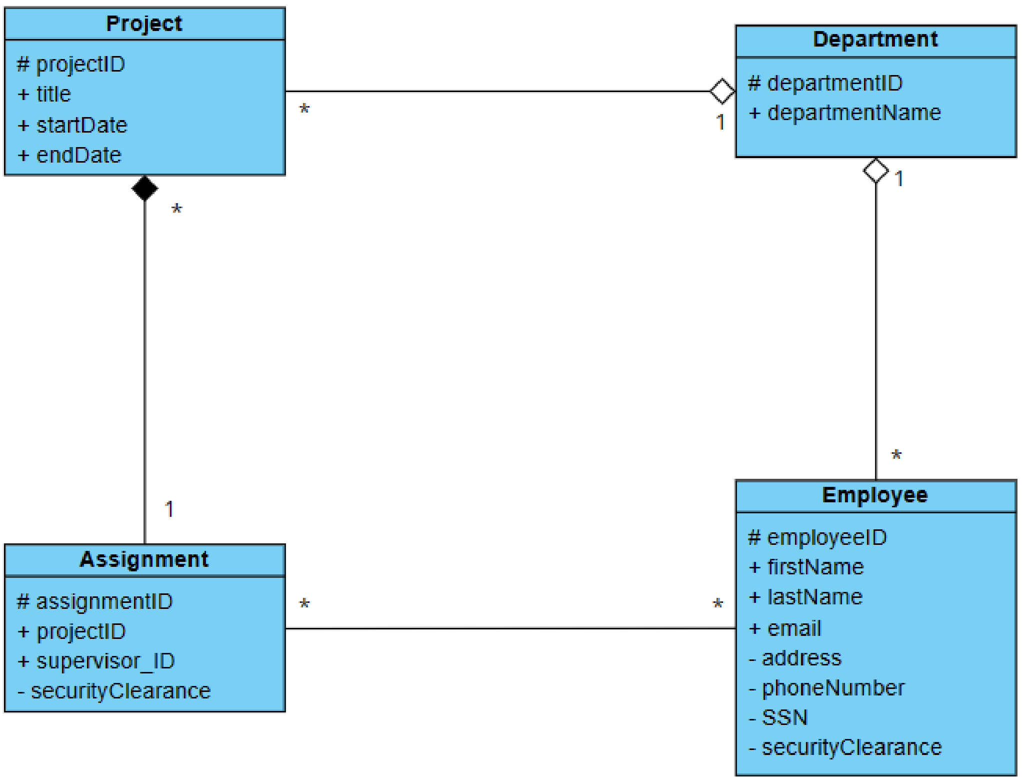


Use case (Second Draft)

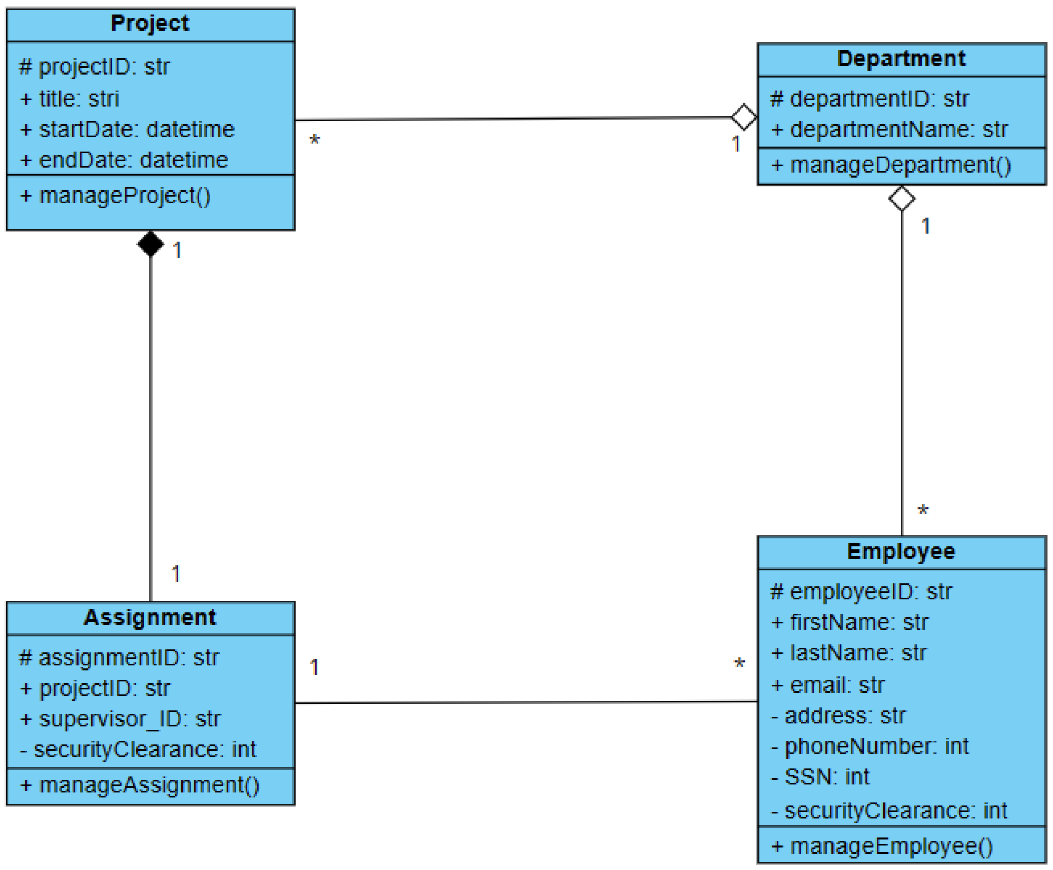


Class Diagram

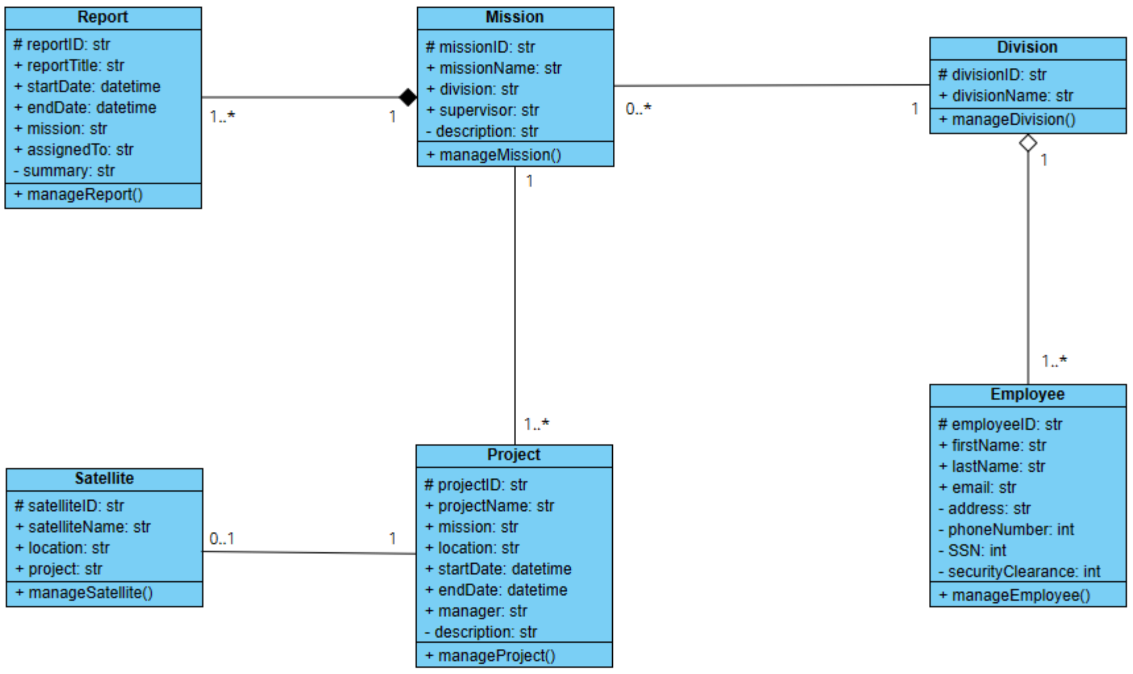
First draft



Second draft


Project
# projectID: str
+ title: stri
+ startDate: datetime
+ endDate: datetime
+ manageProject()
Department
# departmentID: str
+ departmentName: str
+ manageDepartment()
Assignment
# assignmentID: str
+ projectID: str
+ supervisor_ID: str
- securityClearance: int
+ manageAssignment()
Employee
# employeeID: str
+ firstName: str
+ lastName: str
+ email: str
- address: str
- phoneNumber: int
- SSN: int
- securityClearance: int
+ manageEmployee()
*
1
1
1
1
*
1
*

Third draft



References

Morales. (2022) https://lemp.io/the-importance-of-operating-systems-for-nasa-computers/
Ankush (2022) https://geekflare.com/open-source-database/
Brooks, G. (2013) https://digital.gov/2013/04/30/apis-in-government/
Negus, C. (2020) *Linux Bible*: Wiley.
Solarwinds (2022) https://www.loggly.com/ultimate-guide/python-logging-libraries-frameworks/
Pinto & Stuttard (2013) *The Web Application Hacker's Handbook:* Wiley.
Mozilla (2023) https://developer.mozilla.org/en-US/docs/Web/API/HTML_Sanitizer_API
Li, V (2021) *Bug Bounty Bootcamp:* No Starch Press.
https://www.cyberdegrees.org/resources/security-clearances/