**Documentul de specificare a cerințelor – Sistem de Monitorizare și Control Inteligent al Spațiilor de Birouri**

**Introducere**

**Scopul documentului:** Acest document reprezintă specificația cerințelor software (Software Requirements Specification - SRS) pentru aplicația *Sistem de Monitorizare și Control Inteligent al Spațiilor de Birouri*. Documentul are rolul de a defini în mod clar și detaliat cerințele funcționale și non-funcționale ale sistemului, constrângerile de dezvoltare, precum și scenariile de utilizare. El va servi ca bază de comunicare între echipa de dezvoltare, părțile interesate și evaluatori, asigurând o înțelegere comună a obiectivelor și funcționalităților așteptate. Documentul este redactat într-un stil academic formal și oferă suficiente detalii pentru a ghida implementarea și verificarea sistemului.

**Scopul aplicației:** Sistemul de Monitorizare și Control Inteligent al Spațiilor de Birouri (prescurtat SMCISB) are ca scop îmbunătățirea confortului și eficienței mediului de lucru în birouri prin monitorizarea parametrilor de mediu și implicarea activă a utilizatorilor în reglarea acestora. Aplicația permite afișarea în timp real a condițiilor de mediu (temperatură, luminozitate, umiditate, calitatea aerului și nivelul de zgomot) și colectează feedback sub forma voturilor utilizatorilor asupra stării acestor parametri. Prin agregarea feedback-ului și aplicarea unui algoritm inteligent de ajustare, sistemul simulează reglarea automată a condițiilor de mediu, contribuind la confortul ocupanților unui spațiu de birouri.

**Audiența țintă:** Documentul se adresează în principal dezvoltatorilor software implicați în realizarea aplicației SMCISB, managerilor de proiect, precum și profesorilor sau evaluatorilor academici care analizează proiectul. De asemenea, părți interesate pot fi administratorii IT sau managerii de facilități ai unei clădiri de birouri care doresc să înțeleagă modul de funcționare al sistemului. Documentul este redactat într-un limbaj tehnic accesibil pentru cei familiarizați cu dezvoltarea de aplicații în Python și conceptele de monitorizare a mediului interior, dar și clar structurat astfel încât să poată fi parcurs de persoane cu cunoștințe generale în domeniul IT.

**Sfera de aplicare:** Aplicația SMCISB este un sistem stand-alone (independent) care rulează pe stații de lucru locale și nu necesită integrare cu sisteme enterprise sau servicii cloud externe în versiunea sa curentă. Scopul său se aliniază cu eforturile mai largi de creare a spațiilor de lucru inteligente (*smart offices*), unde confortul angajaților și eficiența energetică pot fi optimizate prin tehnologie. Prin implicarea utilizatorilor în procesul de monitorizare, sistemul oferă un beneficiu dublu: (1) colectează percepția subiectivă a confortului în timp real și (2) ajustează în mod inteligent parametrii simulați ai mediului pentru a îmbunătăți confortul, anticipând modul în care un sistem automatizat de climatizare și iluminat ar trebui să reacționeze. Astfel, proiectul oferă o platformă de testare și demonstrare a conceptelor de *clădire inteligentă* la scară mică, cu aplicabilitate în birouri de diferite dimensiuni. Beneficiile așteptate includ creșterea satisfacției ocupanților, identificarea rapidă a condițiilor neconforme (prea cald/rece, prea întunecat etc.) și posibilitatea de a calibra sistemele de control al clădirii pe baza datelor colectate.

**Definiții, acronime și termeni cheie:**

* **SMCISB:** Sistem de Monitorizare și Control Inteligent al Spațiilor de Birouri (aplicația descrisă în acest document).
* **Parametri de mediu:** Măsurători ale condițiilor ambientale din birou, în special temperatură (°C), umiditate (%), luminozitate (nivel de iluminare, în lux), calitatea aerului (indice sau nivel de CO₂/VOC) și zgomot (nivel de sunet, în dB).
* **Valori reale (măsurate):** Valorile parametrilor de mediu obținute direct de la senzori fizici amplasați în spațiul de birouri.
* **Valori simulate:** Valorile interne ale parametrilor, controlate de algoritmul aplicației pe baza voturilor utilizatorilor, care reprezintă starea "țintă" sau simulată a mediului. Aceste valori nu sunt neapărat egale cu valorile reale măsurate, dar pot fi influențate indirect de feedback-ul utilizatorilor.
* **Vot/Feedback:** Opinia unui utilizator privind starea curentă a unui parametru de mediu, exprimată numeric între -3 și +3 (unde valori negative indică că parametrul este perceput ca prea mare, valori pozitive că parametrul este prea mic, iar 0 indică satisfacție sau neutralitate).
* **Algoritm de ajustare:** Logica sistemului prin care valorile simulate ale parametrilor sunt modificate automat în direcția indicată de feedback-ul utilizatorilor, atunci când se acumulează un anumit tipar de voturi (de ex. trei voturi consecutive într-o direcție).
* **Utilizator autenticat:** Persoană care are un cont în aplicație și se conectează cu un nume de utilizator și o parolă pentru a oferi feedback și a accesa funcționalitățile sistemului.
* **Administrator:** (dacă este cazul) Utilizator cu drepturi speciale, care poate gestiona conturile altor utilizatori sau configura anumite setări ale aplicației.

**Referințe:** Pentru elaborarea structurii acestui document s-au avut în vedere standarde și modele de documentație pentru specificarea cerințelor software, adaptate la contextul aplicației de față. Printre referințe se numără șabloanele oferite în cadrul cursului (ex. "Laborator 1 ASIPSI - Documentul de specificare a cerințelor") și bune practici din industrie pentru SRS. Acestea au ghidat organizarea pe capitole și nivelul de detaliu al informațiilor furnizate. Documentul de față este însă centrat specific pe cerințele sistemului SMCISB și nu face trimitere la alte sisteme sau literatură externă, dat fiind caracterul de proiect didactic original.

**Descriere generală**

**Perspectiva produsului**

Aplicația SMCISB se dorește a fi un instrument ușor de utilizat și eficient pentru monitorizarea și controlul inteligent al mediului într-un spațiu de birouri. Din perspectiva produsului, sistemul este unul **stand-alone desktop**, realizat în Python, care rulează local și interacționează cu senzori hardware (dacă sunt disponibili) pentru a citi condițiile de mediu. Nu este o aplicație de tip enterprise distribuit sau bazată pe web, ci mai degrabă un prototip local ce poate funcționa independent pe un calculator personal sau un mini-computer dedicat (ex: un Raspberry Pi conectat la senzori).

În contextul mai larg al sistemelor de clădiri inteligente, SMCISB poate fi văzut ca un subsistem de monitorizare ambientală focalizat pe birouri individuale sau săli de ședințe. El oferă o **interfață grafică** (GUI) prin care utilizatorii din încăpere pot vedea informațiile de la senzori și pot trimite feedback privind confortul lor. Sistemul acționează ca un mediator între utilizatori și mediul înconjurător: colectează percepțiile subiective (prin voturi) și le folosește pentru a determina acțiuni de reglare simulate. Deși în versiunea actuală a produsului aceste acțiuni nu controlează efectiv echipamente (precum termostate, aer condiționat sau lumini), ele pot servi ca bază pentru **viitoare integrări** cu sisteme fizice de automatizare a clădirii. Astfel, produsul demonstrează fezabilitatea conceptului de "birou inteligent adaptiv", unde atât datele obiective (senzorii) cât și preferințele subiective (utilizatorii) contribuie la deciziile de control.

**Funcționalități principale și caracteristici ale sistemului**

La un nivel înalt, SMCISB oferă următoarele funcționalități cheie:

* **Autentificare securizată a utilizatorilor:** Sistemul permite utilizatorilor să se logheze folosind un nume de utilizator și o parolă. Doar utilizatorii autentificați pot trimite feedback, asigurând astfel integritatea datelor (se cunoaște cine a furnizat un anumit vot) și prevenind accesul neautorizat. Există și posibilitatea înregistrării de noi conturi sau adăugării de utilizatori (manual, de către un administrator sau într-o etapă de configurare inițială).
* **Monitorizarea parametrilor de mediu:** Interfața afișează în timp real valorile citite de la senzori pentru fiecare parametru monitorizat (temperatură, umiditate, luminozitate, calitatea aerului și zgomot). Aceste informații sunt prezentate numeric și, acolo unde este relevant, sub o formă grafică intuitivă (de ex. un termometru stilizat pentru temperatură sau o bară de progres pentru nivelul de zgomot). Valorile reale sunt actualizate periodic (de exemplu, la fiecare câteva secunde) pentru a reflecta schimbările din mediu.
* **Feedback-ul utilizatorilor prin voturi:** Pentru fiecare parametru de mediu afișat, utilizatorul autenticat are opțiunea de a vota starea curentă pe o scală de la -3 la +3. Această scală semnifică nivelul de (in)satisfacție: valori negative indică faptul că parametrul este peste nivelul dorit (prea ridicat) și ar trebui **scăzut**, valori pozitive indică faptul că parametrul este sub nivelul dorit (prea scăzut) și ar trebui **crescut**, iar valoarea 0 indică faptul că utilizatorul consideră parametrul în limite confortabile. Voturile utilizatorilor sunt colectate de sistem ca *feedback* asupra condițiilor de mediu.
* **Ajustarea inteligentă a parametrilor (simulare control):** Aplicația include un algoritm simplu de control automat, care acționează asupra valorilor **simulate** ale parametrilor de mediu. Algoritmul analizează feedback-ul primit: atunci când sunt înregistrate **trei voturi consecutive în aceeași direcție** pentru un anumit parametru (trei feedback-uri succesive care indică fie *creștere*, fie *scădere*), sistemul va ajusta valoarea simulată a acelui parametru în direcția respectivă. Spre exemplu, dacă trei voturi la rând indică „prea frig” (voturi pozitive pentru temperatura ambientală), sistemul va mări temperatura simulată cu un pas prestabilit; dacă trei voturi consecutive indică „prea cald” (voturi negative), sistemul va micșora temperatura simulată. Această ajustare este menită să simuleze intervenția unui sistem de control al climei sau iluminatului, bazat pe tendința percepției utilizatorilor. Valorile simulate ajustate vor fi afișate în interfață ca noile setări țintă, însă **este important de menționat că** aceste ajustări simulate **nu influențează direct valorile reale măsurate** – ele reprezintă doar o simulare a ceea ce ar trebui să facă un sistem de control. În practică, pentru prototip, valorile reale ale mediului pot continua să varieze independent (senzorii reflectând condițiile efective), iar sistemul reacționează numai pe baza feedback-ului utilizatorilor, fără a avea control efectiv asupra mediului fizic.
* **Stocarea și istoricul datelor:** Sistemul stochează într-o bază de date locală informații importante pentru funcționare. Acestea includ conturile utilizatorilor (crediențiale de autentificare) și toate feedback-urile (voturile) oferite de utilizatori, cu detalii precum parametrul vizat, valoarea votului și momentul temporar. Stocarea persistă datele între sesiuni de utilizare, permițând analizarea ulterioară a istoricului de feedback sau reluarea valorilor simulate de control de unde au rămas la închiderea aplicației. De asemenea, păstrarea datelor în bază de date asigură integritatea referențială (fiecare vot este legat de un utilizator și un tip de parametru).
* **Interfață intuitivă și prietenoasă:** O atenție deosebită este acordată proiectării interfeței utilizator (UI) folosind biblioteca Tkinter. Interfața este gândită pentru a fi **clară, simplu de navigat și informativă**, chiar și pentru persoane fără cunoștințe tehnice. Elementele de control (butoni, slider-e sau alte componente pentru vot) sunt plasate ergonomic, etichetate sugestiv (ex. “Temperatura actuală: 24°C – Cum o percepi?” cu opțiuni -3 ... +3) și folosesc coduri de culori sau pictograme acolo unde este potrivit (de exemplu, albastru pentru rece, roșu pentru cald, verde pentru starea OK). Aplicația oferă mesaje de confirmare sau notificări atunci când un vot a fost înregistrat și evidențiază vizual când are loc o ajustare automată a unui parametru.
* **Securitatea datelor de autentificare:** Parolele utilizatorilor sunt stocate în format securizat (de ex. hash-uit, nu în clar) în baza de date SQLite, pentru a proteja confidențialitatea. Deși aplicația este una locală, se iau în considerare bune practici de securitate minimală pentru a preveni accesul neautorizat la conturi (ex. verificarea datelor de login, posibilitatea de *log out*, etc.).

**Clase de utilizatori și părți interesate**

În versiunea curentă a sistemului, se disting următoarele categorii de utilizatori:

* **Utilizator obișnuit (angajat în birou):** Persoana care se autentifică în aplicație pentru a consulta parametrii de mediu și a oferi feedback. De regulă, fiecare angajat care folosește aplicația are propriul cont. Utilizatorii obișnuiți pot efectua toate acțiunile normale (login, vizualizare date, votare, logout), dar nu pot modifica setări interne ale aplicației în afara feedback-ului furnizat.
* **Administrator (opțional):** În cazul în care se implementează funcționalități administrative, un utilizator cu rol de administrator ar putea crea conturi noi, șterge sau modifica conturi existente și ar putea reseta parametri sau vizualiza log-uri detaliate ale feedback-ului. Pentru scopul acestui SRS, prezența unui administrator dedicat nu este esențială, dat fiind că aplicația poate fi configurată inițial cu un set de utilizatori. Cu toate acestea, în unele scenarii, echipa tehnică care administrează sistemul clădirii poate avea un astfel de cont administrativ pentru mentenanță.

De asemenea, putem identifica ca părți interesate indirecte **managerii de facilități sau de resurse umane**, care ar fi interesați de rapoarte privind confortul în birouri, însă actuala versiune a aplicației nu oferă rapoarte analitice, fiind centrată pe funcția operațională de reglare în timp real.

**Mediul de operare**

Aplicația este destinată să ruleze pe sisteme de operare **Windows și Linux** (platforme pe care Python 3.10+ și Tkinter sunt disponibile). SMCISB este dezvoltat în Python versiunea 3.10 (sau superioară) și utilizează biblioteca grafică standard **Tkinter** pentru interfața cu utilizatorul. Baza de date este **SQLite**, stocată local ca fișier pe disc. Astfel, mediul de operare trebuie să suporte instalarea Python 3 și să permită operarea unei interfețe grafice (Tkinter funcționează nativ atât pe Windows, cât și pe majoritatea distribuțiilor Linux, cu pachetele adecvate instalate).

Din perspectiva resurselor hardware, cerințele sunt modeste: un computer obișnuit (de exemplu, un PC sau laptop cu cel puțin 4 GB RAM, procesor dual-core) este suficient pentru a rula aplicația. Spațiul pe disc necesar este minimal (câțiva megabyți pentru Python și fișierul de baze de date SQLite). Dacă se utilizează senzori fizici, este necesar ca stația de lucru să fie conectată la aceștia (direct prin porturi USB/serial, sau indirect prin rețea dacă senzorii trimit date prin Wi-Fi). Sistemul nu necesită acces permanent la internet pentru funcționarea de bază, deoarece toate componentele sale (inclusiv stocarea datelor) sunt locale. Internetul ar putea fi necesar doar pentru actualizarea librăriilor Python sau în cazul în care se integrează ulterior un modul de notificare la distanță (ceea ce nu face obiectul acestui SRS).

Este de menționat că aplicația este multiplatformă (Windows/Linux) atâta timp cât sunt disponibile Python și Tkinter. Portarea pe macOS nu a fost testată explicit, dar fiind vorba de Python, este de așteptat că ar funcționa și pe macOS cu ajustări minime, dacă Tkinter este instalat corespunzător. Interfața fiind scrisă în Tkinter, va avea aspectul nativ dat de biblioteca respectivă pe fiecare OS.

**Contextele de utilizare și integrare**

Sistemul SMCISB funcționează cel mai adesea ca o aplicație standalone lansată de un utilizator la începutul zilei de lucru într-un birou. Utilizatorul se autentifică și lasă aplicația deschisă pe un ecran secundar sau minimizată, permițând colectarea continuă a datelor de la senzori și oferirea de feedback oricând este necesar. Într-un scenariu tipic, mai mulți angajați din aceeași încăpere ar putea avea acces la aplicație (fie pe același ecran, logându-se pe rând pentru a-și indica preferințele, fie pe calculatoare separate care însă nu sunt sincronizate în rețea – versiunea actuală nu suportă direct partajarea feedback-ului în timp real între instanțe multiple ale aplicației). De aceea, se recomandă, ca mod de operare, fie:

* **Un singur terminal dedicat în sala de birou**, unde utilizatorii intră pe contul lor și votează (de exemplu, un panou touchscreen montat pe perete, sau un laptop amplasat central).
* **Fiecare utilizator pe propriul dispozitiv, dar votând secvențial:** Dacă doi utilizatori rulează aplicația pe calculatoare diferite în același spațiu, feedback-urile lor nu se vor combina automat decât dacă folosesc aceeași bază de date (posibil prin amplasarea fișierului SQLite pe un folder de rețea). Această configurație nu a fost scopul principal al proiectului, dar este posibilă ca extensie.

Integrări externe actuale: Aplicația nu se integrează încă direct cu niciun serviciu extern sau sistem de management al clădirii. Totuși, designul ia în considerare extensibilitatea – de exemplu, modulul de ajustare automată a parametrilor este izolat logic, astfel încât în viitor, în loc să ajusteze doar o valoare simulată pe ecran, ar putea trimite comenzi către un sistem HVAC real sau către un controler de iluminat (printr-un API specific).

**Constrângeri de proiectare și de implementare (rezumat)**

*(Notă: Constrângerile detaliate sunt tratate în capitolul* [*Constrângeri*](https://chatgpt.com/c/68113089-9fc8-800e-a8db-28ad9fc9d5a7#constr%C3%A2ngeri)*, aici menționăm doar pe scurt principalele aspecte care influențează designul.)*

Arhitectura și implementarea sistemului sunt influențate de câteva decizii de proiectare inițiale:

* Folosirea limbajului **Python** și a **Tkinter** impune o abordare procedurală/simplă pentru UI (Tkinter nu este orientat pe web sau model-vizualizare separat clar, deci logica aplicației și interfața se vor afla în același proces).
* Alegerea **SQLite** ca stocare de date oferă simplitate (fără necesitatea unui server de baze de date separat), însă impune ca scrierea datelor să fie gestionată cu grijă pentru a evita blocări – de exemplu, dacă mai multe instanțe ale aplicației accesează simultan fișierul de DB, se poate crea blocaj.
* Sistemul trebuie să funcționeze **în timp real** din perspectiva utilizatorului; prin urmare, interfața nu trebuie să se blocheze în timpul citirii senzorilor sau scrierii în DB. Aceasta implică utilizarea eventuală a unor fire de execuție separate (threads) sau a mecanismelor asincrone din Tkinter (ex: metoda after() pentru update periodic) pentru a citi senzorii și a actualiza UI fără lag.
* **Ergonomia interfeței:** având în vedere că utilizatorii aplicației sunt persoane fără cunoștințe tehnice avansate, designul a trebuit să sacrifice elemente prea complexe sau date brute greu de interpretat, în favoarea unor reprezentări vizuale intuitive. De exemplu, în loc să se afișeze un indice complex de calitate a aerului brut, se poate afișa un indicator simplu de tip "bun/moderat/scăzut" însoțit de o culoare sugestivă.
* **Timpul de dezvoltare limitat (cadru academic):** Proiectul are un orizont de implementare într-un semestru universitar, ceea ce limitează complexitatea funcționalităților ce pot fi incluse. Multe caracteristici avansate (precum rețea multi-user în timp real, integrare IoT completă, algoritmi de control avansați bazați pe AI) sunt în afara scopului și constituie posibile dezvoltări ulterioare. Prin urmare, soluțiile implementate sunt relativ simple, dar funcționale și ușor de extins ulterior.

În următoarele secțiuni, documentul va detalia cerințele funcționale și non-funcționale ale sistemului, va enunța explicit constrângerile, presupunerile și dependențele, va descrie diferite scenarii de utilizare și va oferi diagrame conceptuale și de arhitectură pentru a ilustra modul în care componentele sistemului interacționează.

**Cerințe funcționale**

Cerințele funcționale descriu în mod clar serviciile, comportamentele și funcțiile pe care sistemul SMCISB trebuie să le furnizeze utilizatorilor. Fiecare cerință funcțională este enunțată astfel încât să fie **clară, verificabilă și necesară** pentru îndeplinirea obiectivelor sistemului. Mai jos este prezentată lista cerințelor funcționale identificate:

1. **Gestionare utilizatori (Autentificare și Înregistrare):** Sistemul trebuie să permită **crearea de conturi noi** pentru utilizatori (prin înregistrare) și **autentificarea utilizatorilor existenți**. Fiecare cont conține cel puțin un nume de utilizator unic și o parolă. Parolele trebuie stocate securizat (hash) în baza de date. Autentificarea are rolul de a preveni accesul neautorizat și de a asocia fiecare feedback cu un utilizator identificabil. După trei încercări eșuate de autentificare, sistemul poate bloca temporar accesul (cerință opțională de securitate). De asemenea, trebuie să existe posibilitatea de **deconectare (logout)** a utilizatorului curent. *(Prioritate: ridicată; Critic pentru securitate)*.
2. **Prezentare date senzori:** Sistemul va afișa, în ecranul principal al aplicației, valorile **curente ale parametrilor de mediu** colectate de la senzori reali. Parametrii monitorizați includ: temperatura aerului, umiditatea relativă, nivelul de luminozitate, calitatea aerului și nivelul de zgomot. Fiecare valoare va fi însoțită de unitatea de măsură corespunzătoare (°C, %, lux, indice calitate aer, dB). Actualizarea acestor valori în UI trebuie să se facă periodic (ex: la fiecare 1-5 secunde) sau ori de câte ori se primește o nouă citire de la senzori. Sistemul **nu trebuie** să înghețe în timpul actualizărilor. Valorile citite sunt doar afișate și înregistrate (dacă este cazul), dar nu vor modifica automat setările sistemului (adică nu influențează direct valorile simulate, conform cerințelor de proiect). *(Prioritate: ridicată; esențial pentru funcția de monitorizare)*.
3. **Interfață de feedback (Votarea parametrilor):** Pentru fiecare parametru de mediu afișat, sistemul va oferi utilizatorului autentificat posibilitatea de a trimite **feedback sub forma unui vot numeric între -3 și +3**. Interfața poate folosi, de exemplu, un slider sau un set de butoane predefinite (-3, -2, -1, 0, +1, +2, +3) lângă afișajul fiecărui parametru. La acționarea controlului (de ex. apăsarea butonului "+1" pentru temperatură), sistemul înregistrează votul utilizatorului pentru parametrul respectiv. Fiecare vot este considerat o cerință de ajustare: un număr negativ înseamnă "parametrul este prea ridicat, scade-l", un număr pozitiv înseamnă "parametrul este prea scăzut, crește-l", iar 0 înseamnă "OK/comfortabil, nu e nevoie de schimbare". Sistemul trebuie să valideze intrarea (de exemplu, să nu permită alte valori în afara intervalului -3...+3 sau dublarea accidentală a votului prin multiple clicuri rapide) și apoi să salveze feedback-ul. *(Prioritate: ridicată; reprezintă funcția centrală de colectare a preferințelor)*.
4. **Înregistrarea feedback-ului în baza de date:** La primirea unui vot de feedback, aplicația trebuie să **salveze datele feedback-ului în baza de date**. Informațiile stocate includ: identificatorul utilizatorului care a votat, tipul parametrului votat (temperatură, umiditate, etc.), valoarea votului (-3..+3) și timestamp-ul (data/ora) la care a fost exprimat feedback-ul. Această cerință asigură păstrarea unui **istoric al feedback-urilor** pentru analize ulterioare sau pentru a permite sistemului să determine tipare (cum ar fi cele trei voturi consecutive). *(Prioritate: ridicată; necesar pentru funcționarea algoritmului de ajustare și audit)*.
5. **Algoritmul de ajustare automată a parametrilor:** Sistemul va implementa un **algoritm de business logic** care rulează de fiecare dată când se adaugă un nou feedback (sau periodic, la un interval scurt), pentru a decide dacă este necesară o ajustare a unui parametru. Cerința specifică este: *dacă există trei voturi consecutive înregistrate care indică aceeași direcție de ajustare pentru un anumit parametru, atunci sistemul ajustează corespunzător valoarea* ***simulată*** *a acelui parametru.* Direcția de ajustare este dată de semnul voturilor:
   * Trei voturi consecutive pozitive (de orice magnitudine +1, +2 sau +3, ordonate cronologic fără voturi negative intercalate) pentru un parametru => **mărire** a valorii simulate a parametrului respectiv (de exemplu creștere cu +1 unitate sau un procent fix).
   * Trei voturi consecutive negative pentru un parametru => **micșorare** a valorii simulate a parametrului (de exemplu scădere cu -1 unitate).

Ajustarea se aplică o singură dată când condiția este îndeplinită, după care contorul de voturi consecutive în acea direcție se resetează (sau se așteaptă următoarele trei voturi similare pentru o nouă ajustare). Sistemul trebuie să reflecte imediat în interfață noua valoare simulată după ajustare, astfel încât utilizatorii să vadă că feedback-ul lor a avut un efect. De asemenea, această valoare ajustată ar trebui să fie reținută intern ca noua țintă sau noul "setpoint" simulat. Este important că **valorile reale ale senzorilor nu se schimbă automat**; dacă mediul nu se modifică fizic, valorile reale pot rămâne nemodificate, ceea ce utilizatorul va observa. Această funcționalitate este deci o simulare a răspunsului unui sistem de control automat. *(Prioritate: medie; funcționalitatea definitorie "inteligentă", dar sistemul de bază funcționează și fără ea, teoretic)*.

1. **Afișarea valorilor simulate ajustate:** Sistemul trebuie să afișeze utilizatorilor nu doar valorile reale de la senzori, ci și **valorile simulate curente ale parametrilor**, cele pe care le ajustează conform feedback-ului. Acestea pot fi evidențiate în interfață alături de valorile reale (de exemplu, "Temperatură: 25.0 °C (reală) / 23.0 °C (țintă simulată)"). Astfel, utilizatorii pot vedea ce țintă urmărește sistemul. Dacă se produce o ajustare automată, noua valoare simulată țintă trebuie actualizată pe ecran. *(Prioritate: medie; sporește transparența sistemului pentru utilizatori)*.
2. **Notificarea utilizatorilor privind ajustările:** Ori de câte ori sistemul aplică o ajustare automată a unui parametru (ca urmare a regulii de trei voturi), ar trebui să existe un **feedback vizual și/sau auditiv** în aplicație care să informeze utilizatorii. De exemplu, un mesaj de tip "Temperatura țintă a fost ajustată la 23°C pe baza feedback-ului primit." ar putea fi afișat pentru câteva secunde, sau parametrul ajustat ar putea fi evidențiat (blink) temporar. Aceasta nu este explicit menționată în enunț, dar reprezintă o cerință derivată pentru a asigura o bună **experiență a utilizatorului** și a-l ține informat. *(Prioritate: scăzută-medie; îmbunătățește UX)*.
3. **Persistența setărilor și reluarea sesiunii:** La închiderea aplicației, sistemul trebuie să salveze starea curentă relevantă astfel încât la următoarea pornire să reia din același punct. Aceasta înseamnă că valorile simulate ale parametrilor ajustate ultima dată trebuie fie salvate (în baza de date sau într-un fișier de configurare), fie recalculabile din istoricul feedback-urilor. O abordare simplă este stocarea valorilor simulate curente în baza de date (de exemplu, în tabelul parametrilor), astfel încât la pornire aplicația să le afișeze direct. Alternativ, dacă istoricul este scurt, ar putea recalcula dacă e cazul (dar stocarea directă e mai simplă). *(Prioritate: medie; util pentru continuitate, dar nu critic dacă se decide reinițializarea de la valori implicite la fiecare pornire)*.
4. **Interfața de administrare (opțional):** Dacă există un rol de administrator, sistemul ar trebui să ofere funcționalități cum ar fi: creare/suspendare ștergere de conturi de utilizator, resetarea parolelor, vizualizarea log-urilor de feedback. Aceasta s-ar putea implementa sub forma unui meniu sau a unei ferestre separate accesibilă doar administratorilor. Pentru versiunea minimă, aceste acțiuni pot fi realizate manual (de ex. inserarea directă în DB a conturilor) și deci cerința poate fi considerată opțională. *(Prioritate: scăzută)*.
5. **Gestionarea erorilor și a situațiilor speciale:** Sistemul trebuie să trateze grațios câteva scenarii de eroare previzibile. De exemplu: dacă senzorii nu furnizează date (conexiune pierdută sau hardware deconectat), aplicația va afișa un mesaj de eroare sau un indicator (ex: "--" în loc de valoare) și va continua să ruleze, eventual încercând reconectarea senzorilor. Dacă baza de date devine inaccesibilă sau coruptă, sistemul va notifica utilizatorul și va încerca să se autorepare (ex: re-crearea fișierului DB, dacă e posibil, cu pierderea datelor non-esențiale). În cazul introducerii unor date invalide (de exemplu, cineva modifică manual fișierul DB și pune date aiurea), aplicația ar trebui să ignore sau să corecteze valorile în afara limitelor așteptate (asigurând de exemplu că voturile sunt întotdeauna între -3 și +3). *(Prioritate: medie; important pentru robusteză, deși unele aspecte țin de implementare)*.

Fiecare cerință de mai sus va fi verificată pe parcursul dezvoltării și testării. Cerințele 1-5 sunt critice pentru ca sistemul să poată atinge obiectivul principal (colectarea feedback-ului și ajustarea parametrilor), în timp ce restul sunt complementare pentru o experiență completă și robustă. Organizarea interfeței și modul de interacțiune sunt strâns legate de aceste cerințe și vor fi detaliate în secțiunea de **Cazuri de utilizare**, unde vom vedea cum un utilizator tipic interacționează cu sistemul pentru a îndeplini aceste funcții.

**Cerințe non-funcționale**

Cerințele non-funcționale descriu proprietățile calitative ale sistemului SMCISB, adică modul în care sistemul furnizează funcționalitățile descrise mai sus. Acestea includ aspecte legate de performanță, utilizare, securitate, fiabilitate, mentenanță etc. Chiar dacă nu adaugă noi funcții, cerințele non-funcționale sunt esențiale pentru satisfacția utilizatorilor și succesul implementării. Mai jos sunt enumerate cerințele non-funcționale relevante pentru SMCISB:

* **Ușurință în utilizare (Usability):** Aplicația trebuie să fie intuitivă și ușor de folosit chiar de la prima interacțiune. Interfața grafică va utiliza termeni clari în limba română și elemente vizuale familiare. Pașii necesari pentru a realiza o acțiune (de ex. autentificare sau trimiterea unui vot) trebuie să fie minimali. Dimensiunea fonturilor și a elementelor UI trebuie să fie suficient de mare pentru a fi lizibile pe un ecran tipic de laptop sau monitor de birou. Un utilizator nou ar trebui să poată înțelege cum să folosească aplicația (cel puțin pentru funcțiile de bază) **fără a consulta un manual**, în maxim câteva minute. De asemenea, culoarea și contrastul în interfață vor fi alese astfel încât să nu obosească privirea în caz de utilizare îndelungată. (Respectarea principiilor de design ergonomic și *user experience* pozitivă este esențială.)
* **Performanță și timp de răspuns:** Sistemul ar trebui să ofere un **timp de răspuns rapid** la acțiunile utilizatorilor. Asta înseamnă că, după ce un utilizator apasă un buton de vot, feedback-ul vizual (ex. un mesaj de confirmare sau actualizarea stării) ar trebui să apară în mai puțin de 0.5 secunde, astfel încât utilizatorul să perceapă interacțiunea ca fiind instantanee. De asemenea, citirea datelor de la senzori și actualizarea UI trebuie să se facă fluent; un ciclu de refresh al datelor de 1 secundă este ideal, dar și unul de până la 5 secunde este acceptabil, atâta timp cât UI-ul nu se blochează. **Performanța algoritmului de ajustare** nu este problematică dat fiind simplitatea lui, însă aplicarea lui nu trebuie să introducă întârzieri vizibile. Aplicația ar trebui să se lanseze complet (timp de pornire) în mai puțin de ~3 secunde pe un hardware modern. Consumul de resurse trebuie să fie redus: memoria RAM utilizată < 100MB, iar încărcarea procesorului să rămână sub 5% în medie (când senzorii transmit date la intervale regulate și userul oferă feedback ocazional).
* **Fiabilitate și disponibilitate:** Având în vedere că aplicația poate rula pe perioade îndelungate (pe durata unei zile de lucru), aceasta trebuie să fie stabilă și să nu sufere de scurgeri de memorie sau blocaje care să necesite restarturi frecvente. Sistemul ar trebui să funcționeze **continuu pe parcursul a cel puțin 8 ore** fără degradarea performanței sau erori. În caz de erori neprevăzute, sistemul ar trebui să **gracefully degrade**: de exemplu, dacă baza de date devine inaccesibilă temporar, aplicația poate continua să afișeze datele și să permită voturi, stocându-le în memorie până la restabilirea conexiunii cu DB. Disponibilitatea sistemului este limitată doar de disponibilitatea stației de lucru pe care rulează, nefiind un serviciu 24/7 critical; totuși, în orele de program, se așteaptă ca aplicația să fie mereu disponibilă pentru utilizatori.
* **Securitate:** Deși este o aplicație locală, securitatea datelor de autentificare și a integrității feedback-urilor este importantă. Cerințe de securitate includ:
  + **Autentificare sigură:** Parolele utilizatorilor trebuie stocate hashed (de ex. folosind un algoritm precum SHA-256 sau bcrypt) pentru a preveni expunerea lor în clar, în cazul accesului neautorizat la fișierul bazei de date.
  + **Autorizare și separare pe conturi:** Doar utilizatorii autentificați pot vota și genera acțiuni; aplicația nu permite accesul la ecranul principal (cu datele senzorilor) decât după login, astfel încât datele să fie vizibile doar de către cei din organizație. (Notă: dacă există un cont comun vizibil pe un panou public, atunci datele senzorilor nu sunt confidențiale, însă tot nu se permit acțiuni fără login).
  + **Protecția datelor stocate:** Baza de date SQLite ar trebui protejată împotriva modificărilor externe neautorizate. De exemplu, fișierul .sqlite poate fi salvat într-un director protejat de permisiuni ale sistemului de operare, astfel încât doar aplicația sau utilizatorul curent să aibă drept de citire/scriere. Nu sunt stocate date personale sensibile (doar username și hash parolă), deci riscul de încălcare a confidențialității este mic.
  + **Securitatea codului:** Aplicația va valida datele de intrare (inclusiv prevenirea injecțiilor SQL chiar dacă SQLite nu e expus la input direct decât prin codul nostru) și va evita executarea de comenzi sistem sau alte practici riscante. Orice canal de comunicare cu senzorii (ex. port serial) trebuie folosit responsabil, pentru a nu permite injecție de comenzi din exterior.
* **Portabilitate și interoperabilitate:** Sistemul trebuie să fie portabil între platforme (cerință deja acoperită – Windows și Linux suportate). Instalarea aplicației ar trebui să fie simplă. De exemplu, se poate furniza ca un pachet executabil *standalone* (folosind PyInstaller sau similar) pentru ușurință, caz în care pachetul trebuie testat pe ambele platforme. Interoperabilitatea cu alte sisteme este limitată deoarece SMCISB este independent, dar codul este scris modular astfel încât să se poată înlocui sursele de date (de exemplu, dacă în locul senzorilor fizici s-ar dori citirea datelor dintr-o altă aplicație sau serviciu, acest lucru să fie posibil prin schimbarea unui modul). De asemenea, datele pot fi exportate manual din DB SQLite, ceea ce oferă un mod de interoperare cu alte instrumente (ex. export CSV al istoricului pentru a-l importa într-un instrument de analiză).
* **Scalabilitate:** În configurația actuală (o instanță locală, câțiva utilizatori, câțiva senzori), scalabilitatea nu este o preocupare majoră. Totuși, dacă ne gândim la extindere, sistemul ar trebui să poată **adăuga ușor noi tipuri de senzori sau parametri** (de exemplu dacă se dorește monitorizarea presiunii atmosferice sau a nivelului de CO₂, structura aplicației să permită adăugarea acestor parametri cu efort minim). Scalarea la mai mulți utilizatori concomitent ar necesita transformarea aplicației într-una de rețea (client-server); acest SRS nu impune așa ceva, însă menționăm că designul actual este scalabil *vertical* (poate rula practic orice număr de feedback-uri în timp, limita fiind puterea mașinii locale), dar nu scalabil *orizontal* fără modificări arhitecturale (nu suportă multi-instance out of the box). Baza de date SQLite suportă câteva conexiuni concurente în modul citire, dar numai una de scriere; pentru volumul nostru mic (un vot la câteva minute), acest aspect este suficient.
* **Mentenabilitate și extensibilitate:** Codul sursă va fi organizat și documentat astfel încât să poată fi ușor de întreținut. Folosirea Python asigură un grad de lizibilitate ridicat. Modulele aplicației (interfața, accesul la senzori, logica de ajustare, accesul la DB) vor fi separate pe cât posibil, permițând modificarea unuia fără a afecta drastic altele. Se vor folosi denumiri intuitive pentru clase și funcții, iar comentariile vor explica părțile mai complexe ale algoritmului (cum ar fi determinarea celor trei voturi consecutive). Extensibilitatea se referă la posibilitatea de a adăuga noi funcții: de exemplu, dacă ulterior se dorește integrarea cu un sistem de notificare pe email când un parametru devine critic, ar trebui să fie relativ ușor de adăugat, datorită structurării pe evenimente (aplicația știe când valorile depășesc un prag sau când primește feedback constant negativ, deci poate lansa un trigger).
* **Teste și calitate:** Fiecare cerință funcțională enumerată va fi asociată cu **criterii de acceptanță** și scenarii de test. De exemplu, un test de acceptanță pentru algoritmul de ajustare: "Simulează trei voturi consecutive pozitive pentru luminozitate și verifică că valoarea simulată a luminozității a crescut după al treilea vot." Sistemul ar trebui să treacă aceste teste. De asemenea, se pot implementa teste unitare pentru funcțiile critice (ex: o funcție care calculează dacă trei voturi sunt consecutive și aceeași direcție).
* **Constrângeri de mediu și siguranță (Safety):** Dacă aplicația ar fi instalată într-un mediu real de birou, ar trebui să nu afecteze negativ infrastructura de rețea sau calculatoarele pe care rulează. Consumul redus de resurse menționat ajută la acest aspect. Nu există implicații de siguranță fizică directă deoarece sistemul nu controlează efectiv echipamente (într-o versiune viitoare, dacă ar controla dispozitive fizice, ar trebui abordate cerințe safety – de ex. să nu supraîncălzească o cameră sau să stingă luminile brusc punând în pericol utilizatorii; în stadiul simulat actual, aceste riscuri nu se aplică).
* **Aspecte de confidențialitate:** Singurele date personale implicate sunt numele de utilizator (care poate fi un username generic, nu neapărat numele real) și preferințele exprimate sub formă de vot. Aceste informații sunt stocate local și nu sunt partajate nicăieri. Totuși, la nivel organizațional, dacă aplicația ar fi folosită pe bune, ar trebui informat personalul că feedback-ul lor este înregistrat (chiar dacă e anonimizat) și eventual agreat, conform politicilor GDPR, dacă se consideră că un vot de confort termic poate fi o dată personală. În context academic restrâns, aceste aspecte sunt minore, dar SRS le menționează pentru completitudine.

În ansamblu, cerințele non-funcționale de mai sus asigură faptul că sistemul nu doar **va face** ceea ce trebuie (cerințele funcționale), dar va face într-un mod care să fie eficient, sigur și plăcut pentru utilizatori. Pe măsură ce proiectul evoluează, aceste cerințe pot servi drept ghid pentru deciziile de implementare și pentru validarea finală a produsului.

**Constrângeri**

În această secțiune sunt enumerate diferite constrângeri care influențează dezvoltarea și funcționarea sistemului SMCISB. Constrângerile pot fi de natură tehnologică, de mediu, de reglementare sau legate de proiect.

* **Tehnologii impuse:** Proiectul este dezvoltat folosind Python 3.10+ și biblioteca GUI Tkinter, având ca stocare baza de date SQLite. Aceste alegeri tehnologice au fost impuse de cadrul didactic al laboratorului (ASIPSI) și nu pot fi schimbate fără a ieși din aria cerințelor de proiect. Astfel, sistemul **nu poate fi web** (nu se va utiliza Django/Flask sau alt framework web) și nu poate folosi un alt SGBD precum MySQL sau PostgreSQL. De asemenea, folosirea Tkinter limitează într-o anumită măsură designul UI comparativ cu tehnologii moderne (nu există control total asupra stilurilor, componente mai complexe trebuie construite manual etc.), ceea ce constituie o constrângere de implementare.
* **Mediul de rulare local:** Aplicația este concepută să ruleze local, pe un singur calculator, nu într-un mediu server-client distribuit. Acest fapt constrânge modul de utilizare – de exemplu, pentru a suporta mai mulți utilizatori simultan în camere diferite ar trebui instanțe separate ale aplicației pe fiecare calculator, iar sincronizarea lor ar fi problematică cu SQLite. Astfel, pentru versiunea curentă, **nu se suportă utilizarea concurentă multi-user pe mai multe stații de lucru** (decât dacă se partajează manual DB, ceea ce nu e recomandat). Această constrângere a mediului local simplifică securitatea (totul e local), dar limitează totodată aria de acțiune a sistemului.
* **Disponibilitatea senzorilor fizici:** Proiectul presupune afișarea valorilor reale de la senzori. Se subînțelege deci că spațiul de birouri vizat trebuie echipat cu senzori de temperatură, umiditate, lumină, calitatea aerului și zgomot, sau alternativ cu un sistem care furnizează aceste valori (fie ele și simulate). O constrângere este **disponibilitatea acestor senzori și integrarea lor cu aplicația Python**. Dacă senzorii sunt conectați prin microcontrolere (ex. Arduino, Raspberry Pi cu senzori), trebuie existent un canal de comunicare (serial, I2C, API web etc.) și driverele/librăriile Python adecvate. În cazul în care nu se dispun de senzori reali la testare, valorile pot fi simulate software (de ex. generând variații random în limite plauzibile), însă cerința aplicației este să fie ușor de conectat la senzori reali dacă aceștia sunt prezenți. Constrângerea hardware este deci: **aplicația funcționează corect numai dacă are sursă de date pentru parametri**. Lipsa unui senzor real pentru un parametru se poate gestiona prin a nu afișa acel parametru sau a-l marca indisponibil.
* **Spațiul de proiectare limitat la parametrii menționați:** Sistemul este proiectat luând în calcul exact cei 5 parametri specificați (temperatură, umiditate, luminozitate, calitate aer, zgomot). Acest cadru fixează tipurile de date, unitățile și chiar aspectul interfeței. Dacă s-ar cere adăugarea unui nou parametru (ex. nivel de CO₂ separat de calitatea aerului), ar fi nevoie de modificări în cod, interfață și bază de date. Constrângerea e că **schema de date și interfața sunt stabilite pe setul dat de parametri** și extinderea necesită efort suplimentar (nu este dinamic configurabil numărul de parametri).
* **Ajustarea la exact trei voturi:** Algoritmul de ajustare e proiectat specific pe criteriul "trei voturi consecutive". Acesta e un **parametru de proiect fix**, nu unul adaptiv. Constrângerea decizională este că sistemul nu va reacționa la mai puține voturi (ex. două nu sunt suficiente) și nu așteaptă mai multe (nu face media pe cinci voturi, de exemplu). Această alegere simplifică implementarea dar limitează finețea controlului. Este deci posibil ca utilizatorii să simtă un ușor decalaj: de exemplu, doi utilizatori votează "-3" (foarte frig) dar până la al treilea nu se întâmplă nimic. Aceasta a fost o decizie luată pentru a evita **oscilațiile bruște** (un singur vot să nu cauzeze imediat reacții, pentru a nu avea comportament de tip "termostat care deschide/clocește mereu"). Constrângerea este deci: **toleranță scăzută la reacție rapidă**, dar mai stabilă. Dacă se dorește alt criteriu (ex. bazat pe medie mobilă sau prag), ar necesita reproiectarea logicii.
* **Lipsa controlului efectiv al mediului:** Așa cum s-a precizat, sistemul simulează controlul, dar nu e conectat la actuatoare reale (de ex. nu trimite semnale către un termostat sau către un sistem de climatizare real). Acesta este un **trade-off** al prototipului: nu s-a urmărit implementarea efectivă hardware de control (care ar fi complicat proiectul). Constrângerea rezultată e că **impactul real asupra mediului este nul** în cadrul de test, ceea ce face dificil de evaluat cum s-ar comporta sistemul într-o buclă de feedback reală. Practic, nu putem valida pe deplin dacă ajustările simulate ar aduce parametrii reali la nivelul dorit, deoarece nu se aplică fizic. Această constrângere se acceptă datorită naturii academice a proiectului, presupunându-se că într-o versiune viitoare s-ar putea conecta la echipamente reale.
* **Interfața unică (o singură fereastră):** Pentru simplitate, aplicația are o interfață principală cu toate elementele. Nu sunt concepute aplicații mobile separate, interfețe web sau multiple ferestre modulare. Constrângerea e că totul se desfășoară în această fereastră, ceea ce poate restrânge complexitatea (nu se vor deschide ferestre pop-up decât eventual dialoguri simple). Aceasta evită complicații de navigare, dar și înseamnă că **toată informația trebuie să încapă** logic într-un singur ecran (scrolling sau tab-uri dacă e nevoie). În implementare, se vor folosi eventual tab-uri sau secțiuni pliabile dacă informația e prea multă.
* **Formatul de stocare al datelor:** Folosind SQLite, toate datele stocate sunt într-un fișier local. Constrângeri implicite:
  + Backup/recuperare: dacă fișierul e șters sau corupt, datele (conturi, istoric) se pierd. Nu există replicare sau backup automat, dat fiind contextul local.
  + Concurență: așa cum s-a menționat, SQLite permite o singură scriere concomitent. Dacă s-ar extinde aplicația și cineva ar monta DB pe rețea, două instanțe ce scriu simultan pot duce la blocări temporare (SQLite lock). În contextul vizat (o instanță), nu e problemă, dar e o limitare de arhitectură.
* **Constrângeri de reglementări:** Nu sunt reglementări specifice cunoscute pentru un astfel de sistem la scara prototip. Dacă ar fi implementat real într-o companie, poate ar exista reglementări privind condițiile de confort (ex. standarde de temperatură, ventilație) – însă aplicația nu controlează direct aceste aspecte, deci nu riscă să violeze vreun standard, se limitează la monitorizare și sugestie. Totuși, dacă datele ar fi considerate personal (feedback-ul), ar intra sub incidența protecției datelor. În context educațional, aceste constrângeri sunt minime.
* **Termen de livrare și resurse:** Fiind un proiect academic, echipa de dezvoltare este formată din studenți, cu timp și buget limitat. Constrângerea temporală (de exemplu, proiectul trebuie finalizat în decursul a ~2 luni, cu predare la un anumit termen) influențează deciziile privind complexitatea - s-au prioritizat cerințele de bază, unele funcționalități au fost simplificate sau marcate ca opționale. De asemenea, nu s-au avut resurse financiare pentru a achiziționa mulți senzori scumpi sau echipamente, astfel încât implementarea s-a realizat cu componente accesibile sau simulări software.

Rezumatul acestor constrângeri arată cadrul în care a fost dezvoltat sistemul. În ciuda numeroaselor constrângeri, acestea au fost luate în considerare în design pentru a asigura că produsul final este realizabil și funcțional în limitele impuse. Orice extindere viitoare a sistemului ar trebui să reevalueze aceste constrângeri (de exemplu, tranziția la o arhitectură distribuită pentru multi-room ar necesita înlocuirea SQLite cu un server de baze de date și a Tkinter cu o interfață web sau aplicații client).

**Presupuneri și dependențe**

În elaborarea cerințelor și designului sistemului SMCISB, au fost făcute o serie de **presupuneri** (asumpții) cu privire la mediul de operare, utilizatori și modul de utilizare, precum și identificate **dependințe** față de componente externe sau condiții care trebuie îndeplinite pentru ca sistemul să funcționeze corect.

**Presupuneri**

* **Disponibilitatea cunoștințelor tehnice pentru instalare:** Se presupune că utilizatorii care vor instala și configura inițial aplicația (de exemplu, personalul IT al companiei sau studentul dezvoltator) au cunoștințe suficiente de operare a unui mediu Python. Cu alte cuvinte, se presupune că Python 3 și pachetele necesare (Tkinter – care e de obicei inclus, eventual librării pentru senzori) sunt instalate corespunzător pe sistemul țintă. Utilizatorul final (angajatul) nu va trebui să facă nimic tehnic, doar să lanseze aplicația și să se autentifice, deci experiența lui nu presupune cunoștințe speciale.
* **Accesul utilizatorilor la un dispozitiv compatibil:** Se prezumă că toți utilizatorii vizați vor avea acces la un PC/laptop sau un dispozitiv dedicat pe care rulează aplicația. Dacă s-a ales varianta unui singur terminal comun în birou, se presupune că acel terminal este plasat într-o zonă accesibilă tuturor (ex. în sala de ședințe sau open-space) și oamenii sunt dispuși să meargă până la el pentru a-și exprima feedback-ul. Alternativ, dacă fiecare are aplicația pe propriul laptop, se presupune că datele de instalare au fost distribuite și fiecare și-a configurat-o local (acest din urmă scenariu, repetăm, nu permite consolidarea ușoară a feedback-ului decât manual).
* **Comportamentul utilizatorilor:** Se presupune că utilizatorii vor oferi feedback **onest și consecvent** despre confortul lor. Adică, dacă le este prea cald, vor vota negativ la temperatura, dacă e prea rece vor vota pozitiv, etc. Se mizează pe ideea că utilizatorii înțeleg scala -3..+3 și o vor folosi în mod rezonabil (nu vor pune +3 când e doar ușor răcoare, rezervând +3 pentru disconfort sever, de exemplu). De asemenea, se presupune că nu vor abuza sistemul introducând în mod repetat voturi contradictorii doar ca să "vadă ce face", deși sistemul trebuie oricum să facă față unor astfel de situații. O altă presupunere este că **nu toți utilizatorii vor vota simultan**; dacă sunt 5 oameni în cameră, probabil 1-2 vor vota inițial și ceilalți doar dacă simt nevoia sau dacă li se cere. Acest lucru e relevant pentru că, dacă toți ar încerca să voteze deodată pe calculatoare diferite, sistemul actual nu e pregătit să facă față coliziunilor (după cum s-a discutat la constrângeri).
* **Stabilitatea mediului fizic:** Se presupune că parametrii reali de mediu nu se schimbă dramatic de la sine într-un interval scurt. De exemplu, temperatura din birou nu va trece de la 20°C la 30°C în 5 minute spontan. Această presupunere asigură că feedback-ul utilizatorilor și reacțiile sistemului pot ține pasul cu schimbările. Dacă ar exista schimbări bruște (ex. se deschide o fereastră iarna și temperatura scade rapid), sistemul ar putea primi multe voturi deodată sau valorile reale ar deveni inconfortabile înainte ca sistemul să fi "ajustat" ceva. Dar, per ansamblu, condițiile într-un birou au inerție și sistemul presupune acest lucru (nu e gândit pentru medii haotice).
* **Calitatea datelor de la senzori:** Se presupune că senzorii folosiți sunt suficient de exacți și calibrați, astfel încât valorile afișate reflectă realitatea. Dacă senzorul de lumină raportează greșit (mult mai jos sau sus decât lumina reală), utilizatorii ar putea oferi feedback nepotrivit deoarece ei percep altceva decât "crede" aplicația. Ne bazăm pe prezumția că senzorii instalați au fost validați și integrați corect. De asemenea, se presupune că datele sosesc la intervale regulate și că latența de citire a senzorilor e mică (câteva secunde cel mult).
* **Autonomia aplicației:** Presupunem că aplicația va rula în modul prevăzut fără intervenție externă. Adică nu există un operator care să stea să ajusteze manual nimic în timp ce rulează. Utilizatorii își dau input-ul (voturi) și restul se întâmplă automat. Această presupunere exclude existența, de exemplu, a unui inginer care stă să citească logurile și să modifice parametri direct în DB sau configurări, în timp real. Totul ar trebui pre-configurat: conturi de utilizator create, eventual valori inițiale setate.

**Dependențe**

* **Dependențe de hardware (senzori și eventual actuatori):** Funcționalitatea de monitorizare depinde de existența senzorilor fizici pentru fiecare parametru sau a unei surse de date echivalente. De exemplu, pentru temperatură și umiditate se poate folosi un senzor DHT22 conectat la un microcontroler; pentru luminozitate un fotorezistor sau senzor de lumină (ex. BH1750); pentru calitatea aerului un senzor de CO₂ sau VOC (ex. MQ-135 sau similar); pentru zgomot un microfon cu circuit de măsurare a nivelului RMS. Sistemul este dependent de driverele sau librăriile care permit citirea acestor senzori în Python. De exemplu, dacă se folosește un microcontroler Arduino, aplicația Python depinde de conexiunea serială cu Arduino și de protocolul de comunicație stabilit (depinde ca Arduino să trimită date într-un format convenit). Dacă un senzor anume nu este disponibil, dezvoltatorul poate decide înlocuirea lui cu o valoare simulată, deci există dependența ca în cod să fie un fallback în lipsa hardware-ului. În viitor, dacă s-ar adăuga actuatori (ex. un releu pentru aer condiționat), aplicația ar depinde de interfața cu acel actuator, dar în versiunea actuală nu e cazul.
* **Dependențe de software extern:** Aplicația depinde de instalarea Python și Tkinter (care vine cu Python, dar pe unele sisteme Linux minim, poate fi nevoie de pachetul python3-tk). De asemenea, dacă anumiți senzori necesită biblioteci speciale (ex. pySerial pentru citire serială, sau biblioteci IoT), atunci există dependența de acele pachete Python. În documentația de implementare va trebui enumerată lista de pachete Python necesare (ex: sqlite3 vine implicit, tkinter implicit, poate requests sau altul dacă se folosește vreo API). Nu se depinde de vreun serviciu cloud sau API online în mod intrinsec (sistemul e local), deci nu e dependent de conexiunea la internet pentru funcțiile sale principale. Cu toate acestea, dacă, de exemplu, s-ar folosi un server NTP pentru timestamp-uri precise sau un server de e-mail pentru notificări, ar exista o dependență de rețeaua externă. Acestea nu sunt însă în cerințele de bază.
* **Dependențe de date inițiale:** Baza de date locală trebuie să conțină anumite date inițiale pentru ca sistemul să fie funcțional. Depindem de existența unui utilizator administrativ sau a unei modalități de a crea conturi. Dacă aplicația se livrează "goală", atunci primul utilizator nu ar putea să se autentifice. O soluție ar fi fie includerea implicită a unui cont "admin/admin" (care ar trebui schimbat apoi parola), fie furnizarea unui script de inițializare a DB. Astfel, există o dependență de rularea acelui script inițial sau de existența unui cont default configurat. De asemenea, se presupune că tabelul parametrilor este populat cu cei 5 parametri și valori implicite (ex. temperatură=22°C, umiditate=50% etc.) pentru pornire. Deci, sistemul depinde de un set de **valori default** raționale. În lipsa acestora, ar porni cu parametri goi și UI-ul nu ar ști ce să arate.
* **Dependențe organizaționale:** Dacă sistemul ar fi folosit într-un context real, ar depinde de suportul managementului pentru a fi adoptat. De exemplu, angajații ar trebui informați despre existența aplicației și îndrumați să o folosească. Această dependență este mai degrabă externă proiectului software în sine, dar contează: fără implicarea utilizatorilor, sistemul nu are feedback și devine inutil. În contextul nostru (proiect academic), dependența se reduce la disponibilitatea evaluatorilor de a-l testa conform scenariilor, deci e mai puțin problematică.
* **Dependența de platforma OS:** Chiar dacă e multiplatformă, ne asigurăm că funcționează pe Windows 10+ și pe o distribuție Linux (Ubuntu 20.04+ de exemplu). Dacă ar fi rulat pe alte medii (ex. un server headless fără interfață grafică, sau un OS exotic), aplicația nu ar funcționa (Tkinter necesită mediu grafic). Deci, depindem de un mediu cu GUI. Totodată, dacă se rulează pe un Raspberry Pi cu Linux, trebuie instalat environment-ul desktop sau folosit modul de afișare pe un mic ecran - e o dependență hardware-software combinată.
* **Dependențe financiare:** Într-un proiect real, costul senzorilor, al unui PC dedicat etc., ar fi o dependență (dacă nu sunt achiziționate, sistemul nu poate fi instalat complet). În contextul nostru, considerăm că echipamentele necesare (sau echivalentele lor simulate) sunt deja la dispoziția echipei.

În concluzie, lista de presupuneri și dependențe evidențiază condițiile în care sistemul va opera corect. În cazul în care oricare dintre aceste presupuneri se dovedește falsă în practică, sau dacă anumite dependențe nu sunt îndeplinite, cerințele sistemului ar putea necesita revizuire. De exemplu, dacă s-ar dori utilizarea simultană de către mulți utilizatori pe dispozitive diferite (ceea ce contrazice presupunerea actuală), atunci soluția tehnică ar trebui regândită (trecere la arhitectură client-server). Documentul consideră însă că, în limitele enunțate, aceste presupuneri sunt rezonabile și dependențele pot fi asigurate.

**Cazuri de utilizare**

Această secțiune descrie principalele cazuri de utilizare (*use cases*) pentru aplicația SMCISB, ilustrând modul în care actorii (utilizatorii) interacționează cu sistemul pentru a-și atinge obiectivele. Actorul principal identificat este **Utilizatorul de birou autenticat**, care poate realiza acțiuni precum autentificarea, vizualizarea datelor și furnizarea de feedback. Pentru completitudine, vom menționa și cazul unui **Administrator** dacă e relevant, deși rolul său este limitat în această versiune. Mai jos sunt prezentate cazurile de utilizare sub formă de diagrame și descrieri textuale.

*Figura 1: Diagrama cazurilor de utilizare ale sistemului, cu actorul principal "Utilizator" și acțiunile sale: autentificare, vizualizare parametri, transmitere feedback, înregistrare cont nou, delogare. (Notă: Ajustarea parametrului este un proces automat inclus în cazul de utilizare de transmitere feedback.)*

**UC1: Autentificare utilizator**

**Actor principal:** Utilizator (neautentificat încă)  
**Scop:** Utilizatorul dorește să se autentifice în aplicație pentru a avea acces la funcțiile de monitorizare și control.  
**Descriere:** La pornirea aplicației, utilizatorul este întâmpinat de ecranul de autentificare. Acesta introduce numele de utilizator și parola în câmpurile corespunzătoare și apasă butonul "Login". Sistemul verifică acreditările comparându-le cu cele stocate în baza de date:

* Dacă datele sunt corecte (exisă un utilizator cu acel username și hash-ul parolei introduse corespunde), accesul este permis. Utilizatorul este redirecționat la ecranul principal al aplicației, unde vede tabloul de bord cu parametrii de mediu.
* Dacă datele sunt incorecte (username inexistent sau parolă greșită), sistemul afișează un mesaj de eroare "Nume de utilizator sau parolă incorectă" și permite reintroducerea. Parola poate fi mascată în UI (de tip \*\*\*\*).
* (Opțional) Dacă există funcția "Remember me" și utilizatorul a selectat-o anterior, poate fi deja autentificat automat la pornire, ocolind acest ecran.

**Condiții prealabile:** Baza de date conține deja contul utilizatorului, iar aplicația este pornită la ecranul de login. Utilizatorul cunoaște numele de utilizator și parola.  
**Postcondiții:** Utilizatorul este autentificat și intră în starea "logat" în sistem, având acum acces la celelalte funcționalități (vizualizare, vot etc.). Sistemul poate înregistra timestamp-ul ultimei autentificări reușite (pentru audit).

*Flux(rezumat din secțiunea anterioară)* Utilizatorul este autentificat cu succes și poate accesa interfața principală.

**UC2: Vizualizare parametri de mediu (Monitorizare)**

**Actor principal:** Utilizator (autentificat)  
**Scop:** Utilizatorul dorește să **vizualizeze starea curentă a mediului** în birou, adică valorile parametrilor monitorizați, în timp real.  
**Descriere:** După autentificare, utilizatorul ajunge pe ecranul principal (dashboard) al aplicației. Aici, el poate vedea afișate valorile curente pentru temperatură, umiditate, luminozitate, calitatea aerului și nivelul de zgomot. Informațiile sunt prezentate numeric, de exemplu: "Temperatură: 24.5°C", "Umiditate: 45%". Pe lângă cifre, pot exista elemente grafice (termometru stilizat, icoană de bec pentru lumină etc.). Utilizatorul pur și simplu privește aceste date. La intervale regulate (ex. o dată pe secundă sau la câteva secunde), valorile se actualizează automat, astfel încât dacă temperatura reală crește la 25.0°C, noua valoare apare fără intervenția utilizatorului. Utilizatorul poate sta pe acest ecran oricât, urmărind cum se schimbă valorile în funcție de mediul real. Dacă un anumit parametru are valori simulată (țintă) diferită, aceasta poate fi și ea afișată (ex. "Țintă: 22°C" la temperatura).

**Condiții prealabile:** Utilizatorul s-a autentificat (UC1). Senzorii sunt conectați și trimit date valide.  
**Postcondiții:** Utilizatorul este informat despre starea curentă a mediului. Nicio modificare de date nu rezultă direct din acest use-case (doar citire).

**Scenarii alternative:** Dacă datele de la un senzor lipsesc (eroare), câmpul respectiv poate afișa "N/A" sau un mesaj de eroare în loc de valoare numerică. Utilizatorul va ști astfel că acel parametru nu este disponibil temporar.

**UC3: Furnizare feedback (vot) pentru un parametru**

**Actor principal:** Utilizator (autentificat)  
**Scop:** Utilizatorul dorește să ofere **feedback** despre confortul său, exprimând un vot privind un anumit parametru de mediu, astfel încât sistemul să poată lua în considerare preferința sa în reglarea ulterioară.  
**Descriere:** Aflat în fața interfeței principale, utilizatorul decide să își exprime părerea despre unul dintre parametri. De exemplu, simte că este prea frig în birou. În dreptul parametrului "Temperatură" din interfață, utilizatorul selectează un vot pozitiv (de exemplu +2). Aceasta semnifică "ar trebui să fie mai cald". După ce utilizatorul apasă butonul sau slider-ul corespunzător valorii +2, sistemul înregistrează instant feedback-ul:

* Aplicația salvează votul în baza de date (creând o nouă înregistrare în tabelul de feedback, legată de utilizatorul curent și parametrul "temperatură", cu valoarea +2 și timestamp-ul curent).
* Interfața poate oferi un **mesaj de confirmare** scurt, de tip "Feedback înregistrat". Utilizatorul vede că acțiunea sa a fost preluată.
* Sistemul declanșează apoi algoritmul de **ajustare automată**. După adăugarea noului vot, aplicația verifică ultimele trei voturi consecutive pentru temperatura camerei. Să presupunem că acesta este al treilea vot consecutiv pozitiv pe temperatura (poate alți doi colegi votaseră și ei "+1" înainte). În acest caz, condiția de ajustare este îndeplinită. **Sistemul ajustează valoarea simulată a temperaturii**: de exemplu crește temperatura țintă cu +1°C peste valoarea anterioară. Imediat, noua valoare țintă simulată apare în interfață (în exemplul nostru, dacă înainte ținta era 22°C, acum va arăta 23°C). De asemenea, sistemul evidențiază această schimbare – de pildă, textul "Temperatura țintă: 23°C" ar putea clipi scurt sau fi evidențiat cu o culoare, iar un mesaj precum "Temperatura a fost ajustată în sus datorită feedback-ului utilizatorilor" poate fi afișat.
* Dacă votul înregistrat **nu** a fost al treilea la rând în aceeași direcție (să zicem că era doar primul sau al doilea), atunci sistemul **nu ajustează încă** nimic, ci așteaptă acumularea de feedback suplimentar. Utilizatorul nu vede nicio schimbare în parametri ca urmare imediată (în afara eventualului mesaj de confirmare).

Utilizatorul poate repeta procesul pentru oricare parametru considerat nepotrivit. De exemplu, dacă i se pare prea întuneric, va oferi un vot pozitiv la luminozitate ("+3" = mult prea întuneric). Aplicația tratează separat feedback-urile pe fiecare parametru (ajustarea la lumină va fi declanșată de trei voturi consecutive pe luminozitate, independent de temperatura).

**Condiții prealabile:** Utilizatorul este autentificat și se află pe ecranul principal (UC1 și UC2 deja efectuate). Parametrii de mediu sunt afișați, deci sistemul este operațional.  
**Postcondiții:** Feedback-ul utilizatorului este salvat persistent. Dacă condițiile sunt întrunite, parametrul vizat este ajustat (simulat) și noua valoare țintă este actualizată în sistem. Utilizatorul va putea vedea, ulterior, istoricul acestor ajustări reflectat în valoarea țintă curentă afișată.

**Scenarii alternative:**

* Dacă utilizatorul își dă seama că a selectat greșit valoarea feedback (ex: voia +1 dar a apăsat +3), el poate furniza imediat un nou vot care să corecteze percepția (sistemul va înregistra un nou feedback; consecutivitatea se calculează în ordine temporală, deci practic ultimul vot contează ca direcție, dar cele anterioare rămân în istoric).
* Dacă baza de date e indisponibilă exact în acel moment (situație rară), sistemul ar putea reține temporar în memorie votul și va încerca să-l salveze ulterior sau va notifica eroarea.
* Dacă trei voturi negative vin după trei voturi pozitive (schimbare bruscă de opinie a utilizatorilor), sistemul nu va acționa până nu are trei consecutive într-o singură direcție; practic se resetează șirul atunci când direcția se rupe.
* Dacă același utilizator votează de mai multe ori la rând același parametru (ex: apasă +2 de trei ori), sistemul va înregistra trei feedback-uri distincte (cu același user și param, dar timestamp diferit). Acestea se consideră tot consecutive și pot declanșa ajustarea. Totuși, interfața ar putea preveni spam-ul de voturi de la un singur utilizator, de exemplu dezactivând butonul pentru câteva secunde după un vot.

**UC4: Delogare (Încheierea sesiunii)**

**Actor principal:** Utilizator (autentificat)  
**Scop:** Utilizatorul dorește să își încheie sesiunea de lucru în aplicație, astfel încât altcineva să nu poată folosi contul său sau aplicația să poată fi închisă în siguranță.  
**Descriere:** La finalul folosirii aplicației (de exemplu, la plecarea acasă sau dacă alt coleg vrea să se logheze), utilizatorul accesează opțiunea de **Logout** (Delogare). Aceasta poate fi un buton dedicat pe interfață sau într-un meniu. După ce apasă Logout, sistemul:

* Îl returnează pe utilizator la ecranul de autentificare, golind datele sesiuni (token-ul de autentificare sau variabila care ține minte utilizatorul logat).
* Oprește eventualele procese legate de utilizator (nu mai acceptă voturi, etc., până la un nou login).
* (Opțional) Dacă s-a implementat salvarea stării la logout, poate marca în DB momentul de logout sau salvează vreo preferință a user-ului.

Aplicația rămâne deschisă la ecranul de login, permițând altui utilizator să se autentifice. Dacă nimeni nu se mai loghează, aplicația poate fi închisă de tot.

**Condiții prealabile:** Utilizatorul este logat (UC1) și se află în orice moment post-autentificare.  
**Postcondiții:** Utilizatorul este delogat; sistemul nu mai consideră niciun utilizator curent logat. Datele vizibile sunt limitate la ecranul de login (nu mai arată parametrii până la următoarea autentificare). Sesiunea fiind închisă, se previne accesul neautorizat la contul respectiv.

**Scenarii alternative:** Dacă utilizatorul închide direct aplicația (fără să apese Logout), efectul este similar - sesiunea se termină implicit. La repornire va cere login. În implementare, s-ar putea recomanda totuși să se apeleze intern rutina de logout la închidere pentru consistență (ex: salvarea eventualelor date nesalvate).

**UC5: Înregistrare cont nou (opțional)**

**Actor principal:** Vizitator (utilizator nou, neautentificat încă)  
**Scop:** Un utilizator care nu are cont își creează unul pentru a putea folosi aplicația.  
**Descriere:** Pe ecranul de autentificare, există o opțiune "Înregistrare cont nou" (Register). Utilizatorul o selectează pentru a deschide formularul de creare cont. El introduce informațiile necesare, precum nume de utilizator dorit și parola (eventual confirmare parolă, email etc., în funcție de implementare). Apasă "Înregistrare". Sistemul validează inputul:

* Verifică dacă numele de utilizator nu este deja folosit. Dacă este deja în DB, întoarce o eroare: "Utilizator existent, alegeți alt nume".
* Verifică dacă parolele respectă politicile (de ex. lungime minimă, conțin cifre etc., dacă s-a impus ceva).
* Dacă totul este în regulă, creează o nouă intrare în tabelul Utilizatori din baza de date, cu username-ul și hash-ul parolei. Poate crea implicit și profilul asociat (de exemplu, dacă ar fi preferințe de utilizator, deși în acest sistem nu sunt).
* Confirmă utilizatorului că înregistrarea a reușit ("Cont creat cu succes").
* În multe aplicații, după înregistrare, utilizatorul fie este logat automat, fie este direcționat înapoi la ecranul de login pentru a se autentifica. În contextul nostru, am putea alege să îl logăm automat ca un aspect practic.

**Condiții prealabile:** Utilizatorul nu are deja un cont. Sistemul este la ecranul de login și vizitatorul selectează opțiunea de înregistrare.  
**Postcondiții:** Un nou cont de utilizator este disponibil în sistem. Utilizatorul poate continua prin a se loga (sau este deja logat dacă s-a prevăzut asta). Baza de date a fost modificată (un utilizator în plus).

**Scenarii alternative:** Dacă există o politică de aprobare a conturilor (de ex. doar administratorul poate crea conturi), atunci opțiunea de înregistrare nu ar fi publică. În acea situație, UC5 nu se aplică, iar conturile noi sunt adăugate separat de un admin (nu avem un use-case detaliat pentru asta aici, dar ar fi un scenariu de administrare). În cadrul proiectului academic, probabil conturile sunt predefinite sau se acceptă autoprovizionarea, astfel încât să se poată testa aplicația ușor.

**Notă:** Cazurile de utilizare de mai sus acoperă interacțiunile principale. Un **Administrator** ar avea cazuri de utilizare separate dacă ar exista interfață pentru el (ex: "Adăugare/ștergere utilizator", "Vizualizare log-uri sistem"). În absența unei interfețe specifice de admin, aceste acțiuni pot fi făcute manual în baza de date, deci nu sunt detaliate ca use-case interactiv.

De asemenea, ajustarea automată a parametrilor nu este un use-case inițiat de un actor uman, ci un **scenariu intern al sistemului** (declanșat de UC3 ca parte a răspunsului sistemului la feedback). Am subliniat acest comportament la UC3. Putem considera totuși un *use case* intern "Ajustare parametru" cu actorul "Timer/Condiție de sistem", dar nu este necesar într-un SRS orientat spre comportamentul vizibil. Important este că utilizatorul vede efectele acestor ajustări ca parte din UC3 (furnizare feedback).

**Modele conceptuale**

Pentru a înțelege structura informațiilor din sistem și relațiile dintre principalele entități, este util un **model conceptual** al domeniului aplicației. În continuare, prezentăm diagrama entităților-cheie ale SMCISB și legăturile dintre ele (practic, un model conceptual ce poate fi transpus ulterior în schema bazei de date):

Figura 1: Model conceptual al bazei de date – entitățile Utilizator, Parametru și Feedback cu relațiile dintre ele.

*Figura 1: Model conceptual al datelor din sistem, incluzând entitățile principale și relațiile dintre ele (Utilizator, Feedback, Parametru).*

Diagrama conceptuală de mai sus evidențiază cele trei entități esențiale ale sistemului și legăturile dintre ele:

* **Utilizator:** reprezintă o persoană care utilizează sistemul (angajatul care se autentifică și oferă feedback). Fiecare utilizator are atribute precum *ID utilizator* (cheie primară unică), *nume de utilizator* (username, unic), *parolă (criptată)*, și eventual *rol* (ex. "admin" sau "user", dacă există diferite niveluri de acces). În modelul nostru, un utilizator poate genera mai multe feedback-uri de-a lungul timpului.
* **Parametru:** reprezintă un anumit aspect monitorizat al mediului de birou (exemplarele: Temperatură, Umiditate, Luminozitate, Calitatea aerului, Zgomot). Atributele includ *ID parametru* (unic, PK), *nume parametru* (ex: "Temperatură"), *valoare simulată curentă* (valoarea pe care sistemul o consideră țintă/curentă pentru acel parametru) și *unitate de măsură* (°C, %, etc.). Numărul de entități Parametru este mic și fix (câte parametri monitorizăm). Fiecare Parametru poate avea asociate mai multe feedback-uri venite de la utilizatori referitor la el.
* **Feedback:** reprezintă opinia exprimată de un utilizator la un moment dat despre un anumit parametru. Este o entitate tranzacțională, legată atât de un Utilizator (autorul feedback-ului) cât și de un Parametru (subiectul feedback-ului). Atribute: *ID feedback* (unic, PK), *utilizatorID* (FK către Utilizatorul care a votat), *parametruID* (FK către Parametrul vizat), *valoare vot* (întreg între -3 și +3) și *timestamp* (data și ora când a fost dat feedback-ul). Fiecare feedback este legat de exact un utilizator și un parametru, iar în timp un utilizator poate genera mai multe feedback-uri (pentru același parametru sau pentru parametri diferiți), la fel cum și un parametru poate avea feedback-uri de la utilizatori diferiți.

**Cardinalități și relații:** Legătura dintre Utilizator și Feedback este de tip **unu-la-mulți**: un utilizator poate avea 0, 1 sau mai multe feedback-uri înregistrate (0 înseamnă că poate există cont fără să fi votat nimic încă), iar fiecare feedback aparține unui singur utilizator. Similar, relația Parametru-Feedback este tot unu-la-mulți: un parametru poate avea mai multe feedback-uri asociate (venite de la diferiți utilizatori și eventual la momente diferite), dar un feedback se referă la un singur parametru. În diagrama conceptuală (Figura 1) acest lucru este indicat prin notațiile de la capetele săgeților: "1" la capătul dinspre Utilizator către Feedback (un utilizator -> mai multe feedback-uri), "0..*" la capătul dinspre Feedback (un utilizator poate să aibă zero sau mai multe feedback-uri). La fel pentru Parametru către Feedback (1 respectiv 0..*).

**Integritatea datelor:** Modelul conceptual sugerează și constrângeri de integritate. De exemplu, nu ar trebui să existe un Feedback înregistrat cu un utilizator inexistent sau un parametru inexistent. Aceasta se va implementa prin chei străine (foreign keys) în baza de date, asigurând integritatea referențială: ștergerea unui Utilizator va duce fie la ștergerea feedback-urilor lui (cascade) fie la interzicerea ștergerii dacă are feedback-uri (depinde de deciziile de implementare), iar ștergerea unui Parametru (dacă ar fi permisă, deși parametrii fiind fixați probabil nu se șterg) ar trebui gestionată similar.

Acest model conceptual servește drept bază pentru proiectarea bazei de date. În continuare, vom detalia structura bazei de date relaționale (SQLite) derivată din acest model.

**Structura bazei de date**

Baza de date a sistemului SMCISB este implementată în SQLite și conține mai multe tabele care corespund entităților din modelul conceptual. Tabelele principale sunt: **Utilizatori**, **Parametri** și **Feedback**. În plus, pot exista tabele auxiliare pentru setări sau jurnale, dar focusul este pe cele trei care stochează datele esențiale. Vom descrie mai jos structura fiecărei mese (tabel), câmpurile și constrângerile acestora:

* **Tabela Utilizatori:** păstrează informațiile despre utilizatorii aplicației.  
  **Coloane:**
  + user\_id – INTEGER, cheie primară (PK), unic pentru fiecare utilizator. Este autoincrement (generat automat la inserare în SQLite).
  + username – TEXT, nume de autentificare al utilizatorului. Trebuie să fie unic (se va impune o constrângere UNIQUE) și ne-null. Exemple: "ion.popescu", "admin", "maria.s".
  + password\_hash – TEXT, șirul de caractere ce reprezintă parola criptată a utilizatorului. Parolele nu se stochează în clar; de exemplu, dacă se folosește algoritmul SHA-256, acest câmp va conține hex- digest-ul de 64 caractere. Este not null (trebuie să existe o parolă).
  + role – TEXT, opțional, rolul utilizatorului (de exemplu "admin" sau "user"). Dacă nu se folosește un mecanism de roluri, poate lipsi sau poate fi NULL/default "user". Acest câmp ar putea fi util dacă dorim să identificăm administratori în aplicație.

**Chei și constrângeri:** Cheia primară este user\_id. Constrângeri: username unic. Pe role se pot defini valori posibile (dacă e cazul, se poate folosi o verificare CHECK (role IN ('admin','user')) de exemplu). Dacă se vrea integritate sporită, se pot adăuga și trigger-e, de pildă un trigger care transformă automat username-ul în litere mici pentru consistență.

* **Tabela Parametri:** reprezintă lista parametrilor de mediu monitorizați de sistem. Aceasta are un număr fix de rânduri (câte parametri se urmăresc) și poate fi populată inițial la crearea bazei de date.  
  **Coloane:**
  + param\_id – INTEGER, cheie primară (PK). Valoare unică pentru fiecare parametru (de exemplu, 1 pentru Temperatură, 2 pentru Umiditate, etc.). Se poate seta autoincrement, deși cum datele sunt puține, se poate și atribui manual.
  + nume – TEXT, denumirea parametrului. Exemplu de date: "Temperatură", "Umiditate", "Luminozitate", "Calitate aer", "Zgomot". Se poate impune UNIQUE pe nume, ca să nu existe două înregistrări cu același parametru. Not null.
  + unitate – TEXT, unitatea de măsură sau indicatorul unității în care este exprimat parametrul. Exemplu: "°C", "%", "lux", "indice", "dB". Acest câmp este mai mult informativ (pentru afișare în UI).
  + valoare\_simulata – REAL (sau numeric), valoarea curentă *simulată* a parametrului, conform ultimei actualizări a algoritmului. Inițial, înainte de orice feedback, aceasta poate fi setată la o valoare implicită (ex: 22.0 pentru temperatura, care ar putea fi considerată confort termic standard). Pe măsură ce sistemul ajustează parametrii, acest câmp se actualizează. Alternativ, s-ar putea alege să nu stocăm aceste valori în DB și să le ținem doar în memorie – dar păstrarea lor în DB ajută să le regăsim la o repornire a aplicației.

**Chei și constrângeri:** param\_id este PK. nume este UNIQUE. Se pot adăuga și constrângeri de tip CHECK pentru a se asigura că valoare\_simulata rămâne într-un interval rezonabil (ex: temperatura să fie între 15 și 30 °C, umiditatea între 0 și 100%). Totuși, astfel de constrângeri pot fi dificil de definit generic și s-ar putea gestiona mai degrabă la nivel de aplicație (prin validare înainte de update).

* **Tabela Feedback:** stochează toate feedback-urile (voturile) trimise de utilizatori. Aceasta va crește potențial nelimitat pe durata folosirii sistemului, conținând fiecare interacțiune de vot.  
  **Coloane:**
  + feedback\_id – INTEGER, cheie primară (PK) pentru fiecare înregistrare de feedback, autogenerată.
  + user\_id – INTEGER, cheie străină (FK) către tabela Utilizatori, indicând cine a dat feedback-ul. Trebuie să corespundă unui user\_id existent în Utilizatori. Dacă se șterge un utilizator, opțiunile sunt: fie ștergem cascadat feedback-urile sale, fie marcăm user\_id cumva (dar mai simplu e cascada). Vom alege constrângerea ON DELETE CASCADE pe acest FK, ca la ștergerea unui cont să se șteargă și feedback-urile (pentru a nu lăsa orfane).
  + param\_id – INTEGER, cheie străină (FK) către tabela Parametri, indicând despre ce parametru este feedback-ul. Trebuie să fie un ID valid din Parametri. Ideal se pune și aici ON DELETE CASCADE sau NO ACTION (dar parametrii oricum nu vor fi șterși uzual).
  + valoare – INTEGER, valoarea votului, în intervalul -3 ... 3. Nu se acceptă valori în afara acestui set. Se poate impune o constrângere CHECK (valoare BETWEEN -3 AND 3). Valorile pot fi și exact -3, -2, -1, 0, 1, 2, 3 (numere întregi).
  + timestamp – DATETIME (în SQLite nu e tip distinct, se poate stoca ca TEXT in format ISO8601 sau INTEGER ca Unix time). Acest câmp indică momentul când utilizatorul a dat feedback. În implementare, la inserție se va completa automat cu CURRENT\_TIMESTAMP sau prin cod cu timpul sistem. Acest câmp este util pentru a determina ordinea voturilor și aplicarea regulii de 3 consecutive, precum și pentru eventuale analize (ex: grafice în timp).

**Chei și constrângeri:** feedback\_id este PK. Chei străine: (user\_id) referință la Utilizatori(user\_id), (param\_id) referință la Parametri(param\_id). Constrângerile ON DELETE: cum menționat, se poate folosi CASCADE pentru ambele, dar în special pentru user. Pentru param, dacă nu intenționăm să ștergem parametri, nu contează. Constrângerea pe valoare asigură domeniul corect. S-ar putea pune chiar și o constrângere ca un anumit utilizator să nu poată avea două feedback-uri identice la același parametru cu același timestamp (dar timestamp-ul oricum diferă dacă se ia până la secundă). Un index compus pe (param\_id, timestamp) ar putea fi util pentru a extrage rapid ultimele feedback-uri la un parametru (util în algoritmul de ajustare), deși la volume mici nu e o problemă.

**Populare inițială a datelor:** Tabelele Utilizatori și Feedback încep goale (cu excepția cazului în care definim un cont de admin implicit). Tabela Parametri va fi populată la inițializare cu cele 5 parametri. Exemplu de conținut inițial al Parametri:

param\_id | nume | unitate | valoare\_simulata

1 | Temperatură | °C | 22.0

2 | Umiditate | % | 50.0

3 | Luminozitate | lux | 300

4 | Calitate aer | ind. | 100

5 | Zgomot | dB | 40

Valorile simulate sunt exemplificative (se pot alege ca medii confortabile implicite).

**Relații și interogări frecvente:** Structura bazei de date astfel creată va suporta interogări precum:

* Selectarea valorilor actuale simulate (din Parametri) pentru a le afișa în UI.
* Actualizarea unei valori simulate într-un Parametri (de exemplu creșterea temperaturii țintă) când algoritmul decide.
* Inserarea unui nou Feedback când un utilizator votează.
* Interogarea ultimelor 3 feedback-uri pentru un anumit param\_id (ordonate descendent după timestamp) pentru a verifica dacă sunt toate de aceeași semn. Aceasta poate fi realizată printr-o interogare SQL cu LIMIT, sau la nivel de cod preluând toate feedback-urile recente.
* Gestionarea utilizatorilor: inserarea la înregistrare, validarea la login (o interogare SELECT pe Utilizatori unde username=... și verificarea hash-ului).

**Considerații de securitate a datelor:** Fiind SQLite, nu există utilizatori separați la nivel de DB. Securitatea se bazează pe restricțiile de acces la fișierul .db. Ar fi indicat ca fișierul bazei de date să fie stocat într-un folder protejat (sau criptat la nivel de disc, dacă e cazul), mai ales pentru a feri parolele (chiar dacă sunt hash) de actori rău intenționați. Parolele sunt oricum hash, deci compromise ar fi greu de inversat, dar e bună practică. O altă opțiune ar fi folosirea unei fraze secrete de criptare a bazei SQLite, însă asta complică lucrurile și implică stocarea cheii în aplicație (ceea ce nu aduce mare securitate suplimentară într-un scenariu local).

În concluzie, baza de date este relativ simplă, cu doar trei tabele majore. Această structură realizează maparea directă a modelului conceptual în model logic relațional, permițând stocarea tuturor informațiilor necesare (useri, parametri, feedback-uri) cu integritatea datelor asigurată prin legături și constrângeri.

**Alte diagrame și explicații suplimentare**

Pentru a completa înțelegerea sistemului, prezentăm diagrama de arhitectură a soluției și discutăm câteva aspecte de implementare și fluxuri interne. Diagrama de arhitectură oferă o vedere de ansamblu asupra componentelor principale (module software și elemente hardware) și a interacțiunilor dintre ele.

*Figura 2: Diagrama de arhitectură a sistemului, ilustrând interacțiunile dintre utilizator, interfața Tkinter, logica aplicației, senzorii de mediu și baza de date SQLite.*

În Figura 2, putem observa următoarele componente și interacțiuni:

* **Utilizatorul** (actor extern) interacționează cu **Interfața Utilizator (UI)** a aplicației, reprezentată de fereastra și elementele grafice Tkinter. Săgeata de la Utilizator la UI (“Acțiuni utilizator (vot, login)”) indică faptul că utilizatorul introduce date sau comenzi: ex. tastează credentiale, apasă butoane de vot, etc.
* **Interfața Utilizator (Tkinter GUI)** trimite aceste **comenzi și date** către **Logica aplicației**. De exemplu, la login, UI transmite modulului de logică numele și parola introduse pentru verificare; la un vot, trimite valoarea votului și parametrul selectat.
* **Logica aplicației** este nucleul programului, care conține implementarea regulilor de business: verificarea autentificării, procesarea feedback-urilor, aplicarea algoritmului de ajustare. Modulul de logică:
  + Primește intrări de la UI și decide cum să reacționeze (de exemplu, la login validează contul consultând baza de date; la un vot, scrie în DB și verifică dacă trebuie ajustat un parametru).
  + Citește date de la **senzorii de mediu** (componenta “Senzori de mediu” din diagramă). Săgeata “Valori senzori (citire)” indică fluxul de date brute de la senzori către logică. Această citire poate fi periodică (ex: o funcție în logică ce interoghează senzorii la fiecare X secunde) sau pe bază de eveniment (dacă senzorii trimit date).
  + Interacționează cu **Baza de Date (SQLite)** atât pentru a **interoga/stoca date** (săgeata către DB “Interogări/Stocare date”) cât și pentru a primi date (săgeata dinspre DB “Date utilizatori/feedback”). De exemplu, logica interoghează DB la autentificare (select user), inserează un nou feedback când primește un vot (insert), actualizează un parametru când se face o ajustare (update), sau preia valorile simulate curente la pornirea aplicației (select).
  + Trimite actualizări către UI (săgeata “Actualizări interfață” dinspre Logică spre UI). Acestea includ: afișarea valorilor curente de la senzori (pe măsură ce sunt citite), afișarea mesajelor (eroare la login, confirmare la vot), actualizarea valorilor simulate după o ajustare automată, etc. Practic, UI-ul se reînnoiește pe baza deciziilor logicii.
* **Baza de Date SQLite** stochează permanent informațiile despre utilizatori și feedback, și temporar (persistă între sesiuni) valorile simulate ale parametrilor. Logica aplicației se ocupă de toate interogările DB necesare, UI-ul neinteracționând direct cu DB-ul. În diagrama, interacțiunea DB este exclusiv cu modulul de Logică.
* **Senzorii de mediu** (dacă sunt prezenți fizic) constituie sursa datelor reale pentru parametri. Ei pot fi, de exemplu: un termometru digital, un higrometru, un senzor de lumină conectat la microcontroller, etc. Aplicația trebuie să comunice cu acești senzori, fie direct (prin porturi seriale/USB, biblioteci hardware) fie printr-un serviciu intermediar. În diagrama am ilustrat comunicarea directă (Logica citește valorile), deși implementarea exactă poate varia. Dacă nu există senzori reali, această componentă poate fi înlocuită de un modul de simulare care generează valori. Indiferent de sursă, modulul Logică tratează aceste date uniform.

**Fluxul fizic al proceselor:** În modul de operare, aplicația urmează un anumit flux de inițializare și rulare:

1. **Pornirea aplicației:** La lansare, se inițializează conexiunea la baza de date SQLite (se deschide fișierul DB, se verifică sau creează tabelele dacă e prima rulare). Apoi, se creează fereastra principală Tkinter. Se pornește eventual un fir de execuție secundar sau un timer periodic pentru citirea senzorilor (pentru a nu bloca UI-ul).
2. **Autentificare:** Utilizatorul vede ecranul de login. Când introduce datele și confirmă, UI-ul trimite datele modulului Logică. Logica face un query la DB pentru acel username. Dacă găsește, compară hash-ul parolei. Rezultatul (succes/eroare) e trimis înapoi la UI pentru a continua fluxul (UI-ul va trece la ecranul principal dacă e succes, sau va afișa eroare și va aștepta reintroducere).
3. **Încărcare date inițiale în UI:** Odată autentificat, UI-ul principal afișează parametrii. Logica, la construire, va fi populat deja valorile simulate actuale (din DB, tabela Parametri) și le transmite UI-ului pentru afișare. UI-ul poate afișa și valori reale curente (dacă citirea senzorilor a furnizat deja vreun set la momentul respectiv; altfel în câteva secunde vor veni).
4. **Bucla principală de monitorizare:** În timpul cât aplicația e deschisă și un utilizator e logat, se desfășoară în paralel două activități principale:
   * **Citirea periodică a senzorilor:** Modulul Logică interoghează senzorii la interval (sau primește notificări de la senzori). Când obține noi valori (ex: temperatura actuală 24.7°C), actualizează UI-ul corespunzător (eticheta de temperatură reală). Valorile reale nu influențează direct altceva (nu schimbă valori simulate), deci sunt doar afișate.
   * **Interacțiunea utilizatorului:** Utilizatorul poate, la orice moment, să ofere feedback folosind UI-ul. Când apasă un buton de vot, UI-ul trimite evenimentul modulului Logică. Logica procesează: inserează feedback-ul în DB, apoi verifică regulile. În mod practic, logica poate ține într-o structură proprie ultimele 2 feedback-uri pentru fiecare parametru și, când vine al 3-lea, detectează secvența. Sau poate interoga DB-ul: "SELECT valoare FROM Feedback WHERE param\_id=X ORDER BY timestamp DESC LIMIT 3" și vede dacă cele 3 sunt toate >0 sau <0. Să presupunem că trei sunt pozitive: logica va calcula noua valoare simulată (ex: valoare\_simulata = valoare\_simulata + increment). Incrementul poate fi constant (de ex. +1 unitate) sau poate ține cont de cât de mari au fost voturile (de ex, dacă toate trei au fost +3, poate crește cu 2 unități; dacă au fost slabe +1, +1, +1, poate crește cu 0.5 – aceste detalii de algoritm pot varia; versiunea simplă e pas fix).
   * După calcul, logica face un UPDATE în tabela Parametri pentru param\_id-ul respectiv cu noua valoare\_simulata. Apoi trimite UI-ului informația că parametru X și-a schimbat ținta. UI-ul atunci actualizează afișajul (ex: schimbă eticheta "Țintă" a acelui parametru). De asemenea, UI poate afișa un mesaj temporar de notificare a ajustării.
5. **Situații de eroare:** Dacă un senzor devine indisponibil, modulul de Logică poate detecta excepția la citire (ex: nu răspunde) și va notifica UI-ul să afișeze "N/A" cum am menționat. Aceasta se face prin fluxul Logică -> UI, similar cu actualizările de valori normale, dar cu un flag de eroare.
   * Dacă apar erori la scrierea în DB (spațiu pe disc epuizat, fișierul devine read-only etc.), logica va captura eroarea și poate anunța UI-ul (poate afișa un pop-up "Eroare salvare date!").
   * Eroarea "username sau parolă greșită" este deja menționată la login (flux UI -> Logică -> UI).
6. **Închidere aplicație:** Când utilizatorul se deloghează sau închide fereastra, modulul de Logică poate opri thread-ul de senzori. Orice date nesalvate se scriu (de obicei toate erau deja salvate în timp real). Aplicația se închide grațios.

**Detalii de implementare:**

* *Concurența și sincronizarea:* Tkinter operează pe un singur thread (mainloop-ul UI). Dacă folosim thread separat pentru senzori, orice actualizare UI din acel thread trebuie sincronizată (Tkinter nu e thread-safe). De obicei se folosește after() pe UI thread: thread-ul de senzori pune valorile într-o coadă, iar UI-ul la fiecare X ms verifică coada și actualizează. Astfel, diagrama arată conceptual actualizarea, dar din punct de vedere implementare se respectă constrângerile bibliotecii.
* *Algoritmul de ajustare:* Este relativ simplu în concept (3 voturi la rând -> ajustare). În cod, se va implementa poate sub forma:
* feedbacks = get\_last\_feedbacks(param\_id, limit=3)
* if len(feedbacks) == 3:
* if feedbacks[0].valoare > 0 and feedbacks[1].valoare > 0 and feedbacks[2].valoare > 0:
* param.valoare\_simulata += PAS\_AJUSTARE
* elif feedbacks[0].valoare < 0 and feedbacks[1].valoare < 0 and feedbacks[2].valoare < 0:
* param.valoare\_simulata -= PAS\_AJUSTARE
* # altfel, nu face nimic

Astfel de cod ar rula după fiecare inserare de feedback. PAS\_AJUSTARE poate fi o constantă (ex: 1 unitate pentru temperatură, sau 5% pentru umiditate, etc., eventual personalizat per parametru).

* *Extensibilitate:* Arhitectura prezentată permite extinderi. De exemplu, dacă s-ar dori trimiterea de notificări email când confortul e foarte scăzut: modulul Logică ar fi locul de a integra un apel la un serviciu de mail atunci când detectează repetitiv voturi extreme sau parametri în afara plajei normale. Un alt exemplu, dacă s-ar conecta la un sistem HVAC real: modulul Logică, imediat ce ajustează valoarea\_simulata, ar putea și să trimită un semnal către termostat (prin API sau protocol Modbus etc.) pentru a schimba efectiv temperatura. UI-ul și DB-ul ar rămâne la fel, deci restul sistemului e pregătit pentru o astfel de integrare cu efort suplimentar moderat.
* *Jurnalizare:* Sistemul ar putea înregistra evenimente notabile (autentificări, ajustări) într-un log (fie fișier text, fie într-o tabelă nouă, ex: LogEvenimente cu timestamp, user, acțiune). Acest lucru nu a fost cerut explicit, dar e o practică bună. În context academic, putem considera că istoricul feedback-urilor ține loc de jurnal, deoarece oricum acolo se vede cine ce a votat și implicit când s-au făcut ajustări (dacă examinăm secvențele din DB).
* *Interfața cu utilizatorul:* Deși SRS nu intră în detalii de design UI, merită menționat că ferestrele vor fi structurate probabil astfel:
  + Fereastra Login: două câmpuri (user, parolă) și buton Login (și eventual buton Inregistrare).
  + Fereastra Principală: afișează parametrii într-o formă tabelară sau card-uri, fiecare cu: nume parametru, valoare reală actuală (senzor), unitate, valoare țintă (simulată), control de vot (de la -3 la +3). Jos, eventual un buton Logout. Poate și eticheta "Autentificat ca: X".
  + Ferestre de dialog mici pentru erori sau confirmări.
  + Schema culori: parametrii ar putea fi colorați în funcție de situație (ex: dacă valoarea reală diferă mult de țintă, poate afișăm cu alt fundal), dar asta e o decizie de implementare.

În ansamblu, arhitectura este una tipică pentru o aplicație desktop standalone: avem straturi de **interfață** (Tkinter), **logică de business** (cod Python care face legătura între UI, DB și senzori, implementând reguli specifice) și **stocare** (SQLite, plus intrările de la senzori ca "input extern"). Toate aceste componente rulează pe aceeași mașină, comunicând direct (nu prin rețea). Această simplitate corespunde scopului proiectului și cerințelor inițiale.

***Încheiere:*** Documentul de față a detaliat specificațiile de cerințe pentru Sistemul de Monitorizare și Control Inteligent al Spațiilor de Birouri. Au fost acoperite aspectele funcționale (ce trebuie să facă sistemul, cum interacționează utilizatorii cu el) și cele non-funcționale (calități și constrângeri), s-au prezentat modele conceptuale ale datelor, structura bazei de date și diagrame de arhitectură. Aceste specificații vor ghida implementarea sistemului, asigurând că produsul final corespunde așteptărilor și cerințelor. Pe parcursul dezvoltării, documentul SRS poate servi ca referință și poate fi actualizat pe măsură ce apar detalii noi sau schimbări în design. Scopul final este realizarea unei aplicații funcționale, ușor de utilizat, care să demonstreze conceptul de feedback al utilizatorilor integrat în controlul inteligent al mediului de birou.