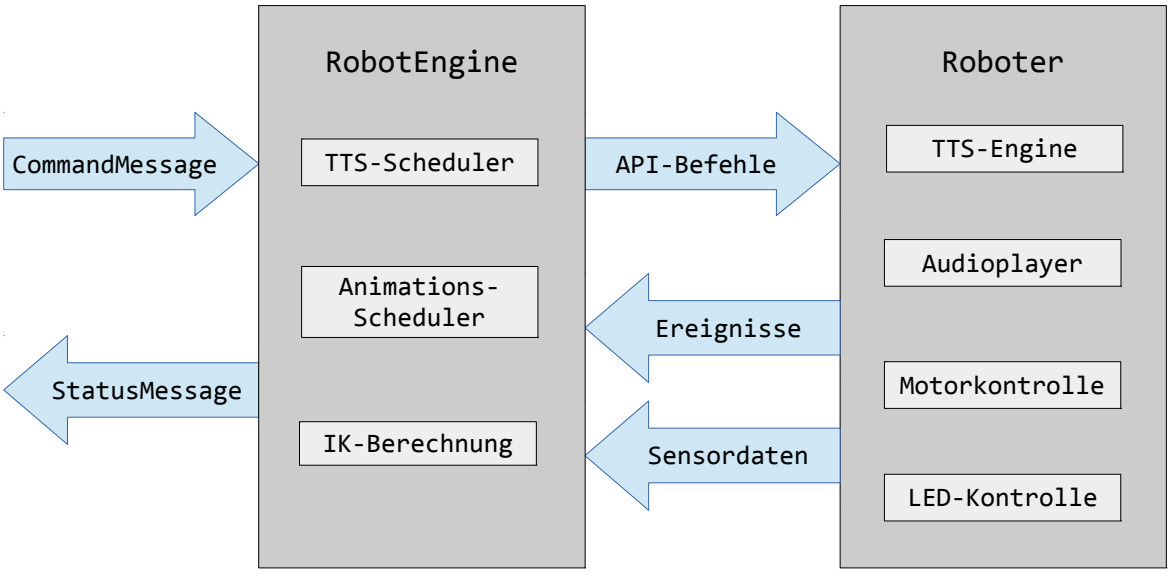
Inhaltsverzeichnis

Wie sieht das Nachrichtenprotokoll aus?
Wie verwendet man eine RobotEngine?
Wie verwendet man die DefaultControlApplication?
Wie implementiert man eine RobotEngine?
Wie implementiert man eine Kontroll-Anwendung?

Wie sieht das Nachrichten-Protokoll aus?



- Grundprinzipien:
  - Datenfluss:
    - *RobotEngine* erhält ein Aktionskommando von außen
    - *RobotEngine* versucht, das Kommando auszuführen
    - *RobotEngine* schickt Status-Meldungen nach außen, um den Fortschritt der Ausführung anzuzeigen
  - Datenformat:
    - Task-ID ermöglicht eindeutige Zuordnung zwischen dem Kommando und den zugehörigen Status-Meldungen
    - beide Nachrichten-Typen besitzen eine kanonische XML-Repräsentation

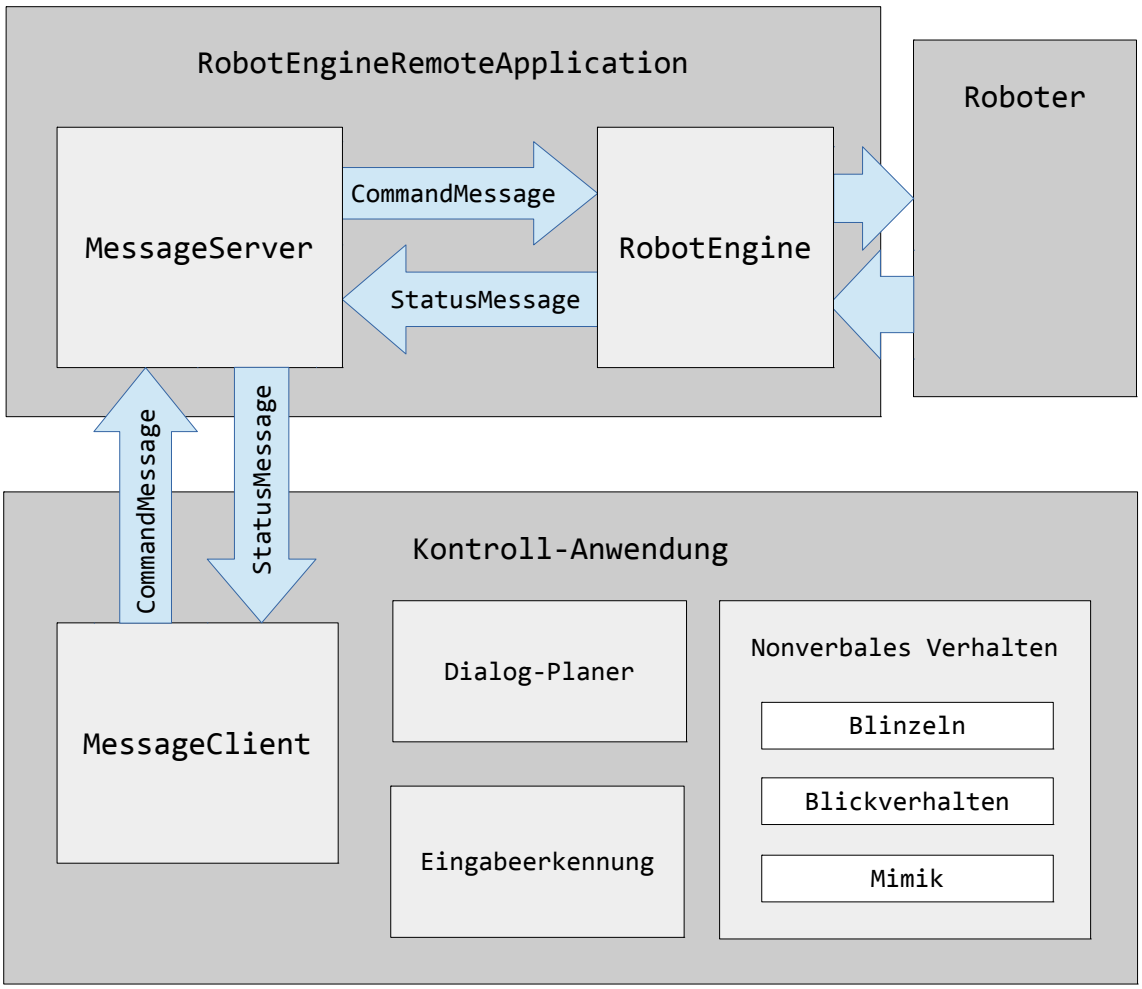
- Paket *de.hcm.robots.messaging*
  - Container-Klassen für ein- und ausgehende Nachrichten
  - Klassen *MessageServer* und *MessageClient* für den Nachrichten-Austausch über UDP
  - Interfaces für Klassen, welche die jeweiligen Nachrichten-Typen verarbeiten

- eingehende Aktions-Kommandos: Klasse *CommandMessage*
  - Eigenschaften:
    - eindeutige Task-ID
    - Typ der Aktion, z.B. “anim” oder “speech”
    - Liste beliebiger Parameter zur Ausführung der Aktion
  - Beispiele:
    - `<command task="N6::Sample_Anim::t0::u0" type="speech" text="Jetzt guck ich mal \\book=0 fröhlich."/>`
    - `<command task="N6::Sample_Anim::t0::u0::b0" type="anim" name="Emotions/happy"/>`
    - `<command task="N6::Sample_LED::t1::u0::b0" type="led" color="yellow" side="left"/>`

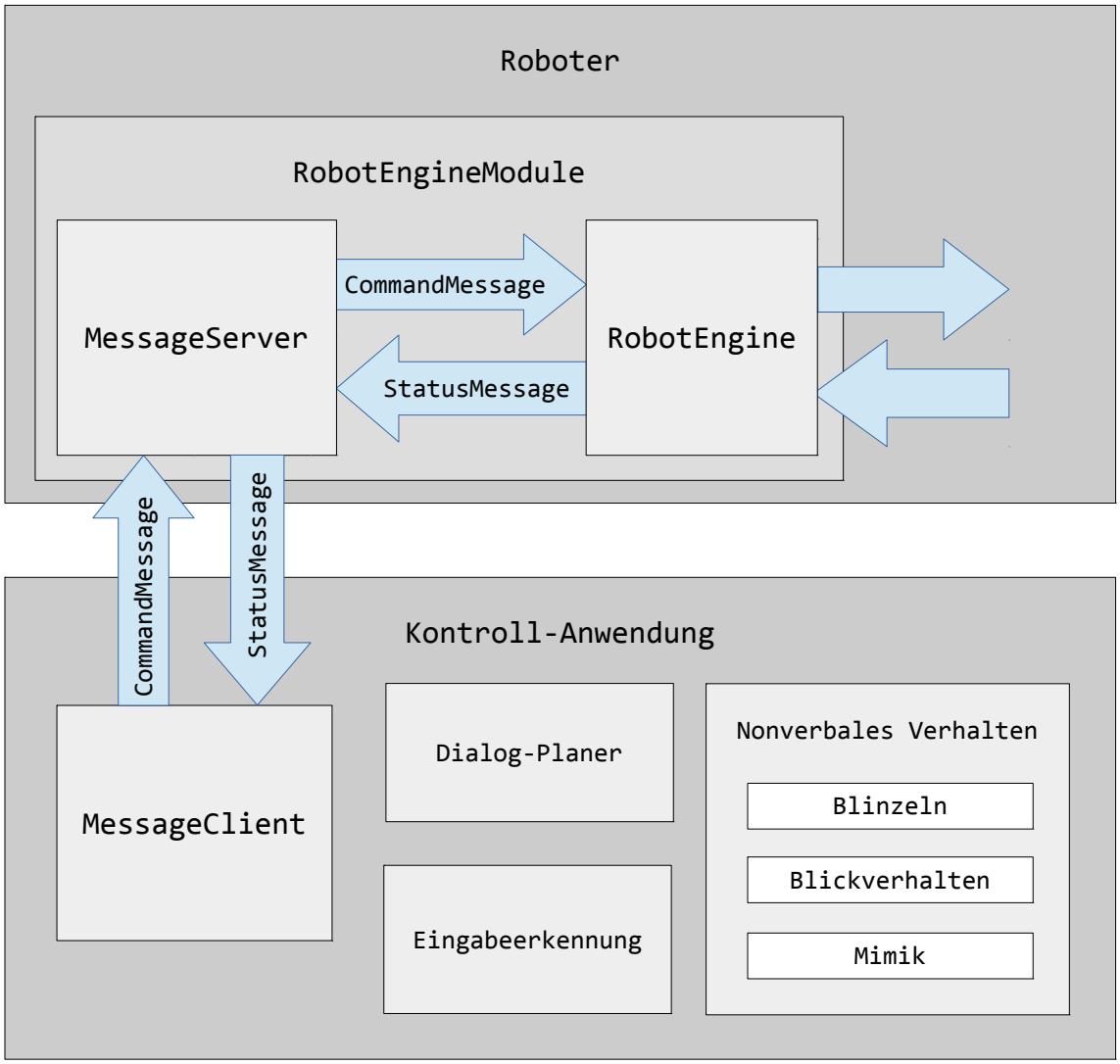
- ausgehende Status-Meldungen: Klasse *StatusMessage*
  - Eigenschaften:
    - Task-ID des zugehörigen Kommandos
    - aktueller Status der Kommando-Ausführung, z.B. “finished” oder “bookmark”
    - Liste beliebiger Details zum Ausführungs-Status
  - Beispiele:
    - <status task="N6::Sample_Anim::t0::u0" status="bookmark" id="0"/>
    - <status task="N6::Sample_Anim::t0::u0::b0" status="finished"/>
    - <status task="N2::Sample_Reset::t1::u0::b0" status="rejected" reason="file not found"/>

Wie verwendet man eine RobotEngine?

- Variante A: UDP-Verbindung zu externer Kontroll-Anwendung
  - z.B. Start über *de.hcm.robots.RobotEngineRemoteApplication*
    - Startargument: Pfad zur Konfigurationsdatei (kompatibel mit *java.util.Properties*)
    - Konfigurations-Parameter:
      - *engine.class*: vollständiger Klassen-Name der verwendeten RobotEngine
      - *engine.config*: Pfad zur Konfigurations-Datei für die verwendete RobotEngine
      - *network.localIP* und *network.localPort*: Adresse, auf welcher der UDP-Server Kommandos empfängt
    - startet einen *MessageServer* zum Empfang der Kommandos
    - akzeptiert außerdem Kommandos über Konsolen-Eingabe (XML-Format)

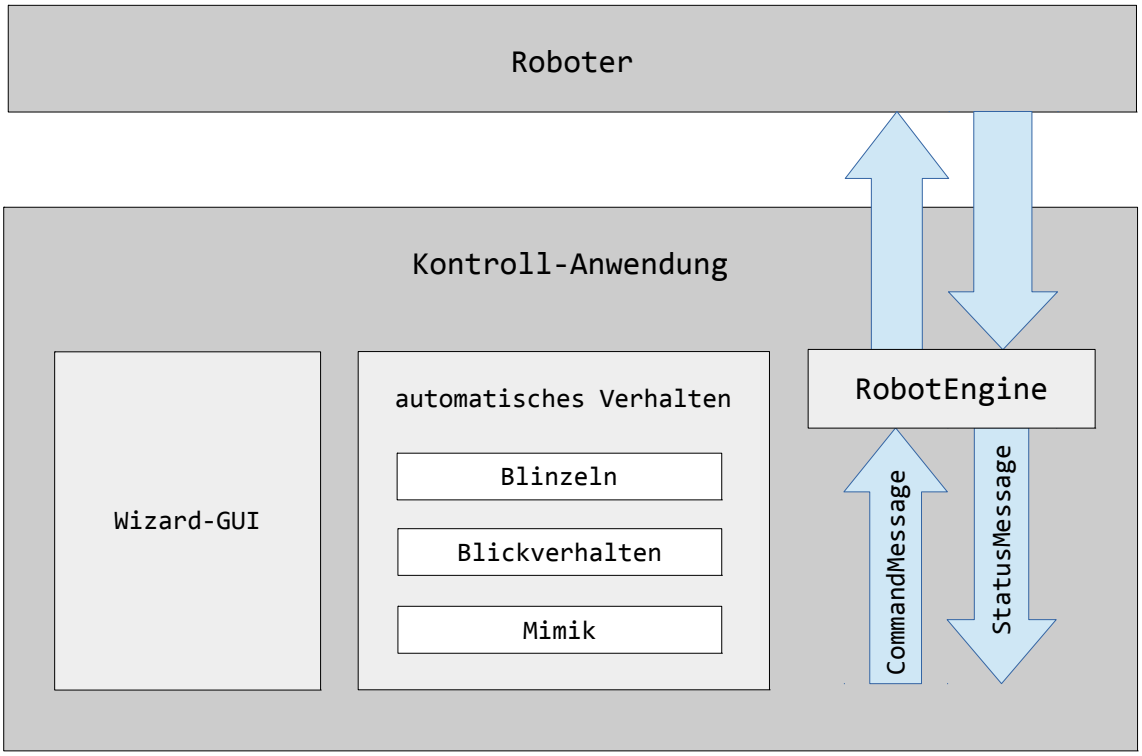


- z.B. als lokales Modul direkt auf dem Roboter
  - idealerweise so eingerichtet, dass es automatisch geladen und gestartet wird
  - konkretes Beispiel: Urbi-Modul auf Reeti V1

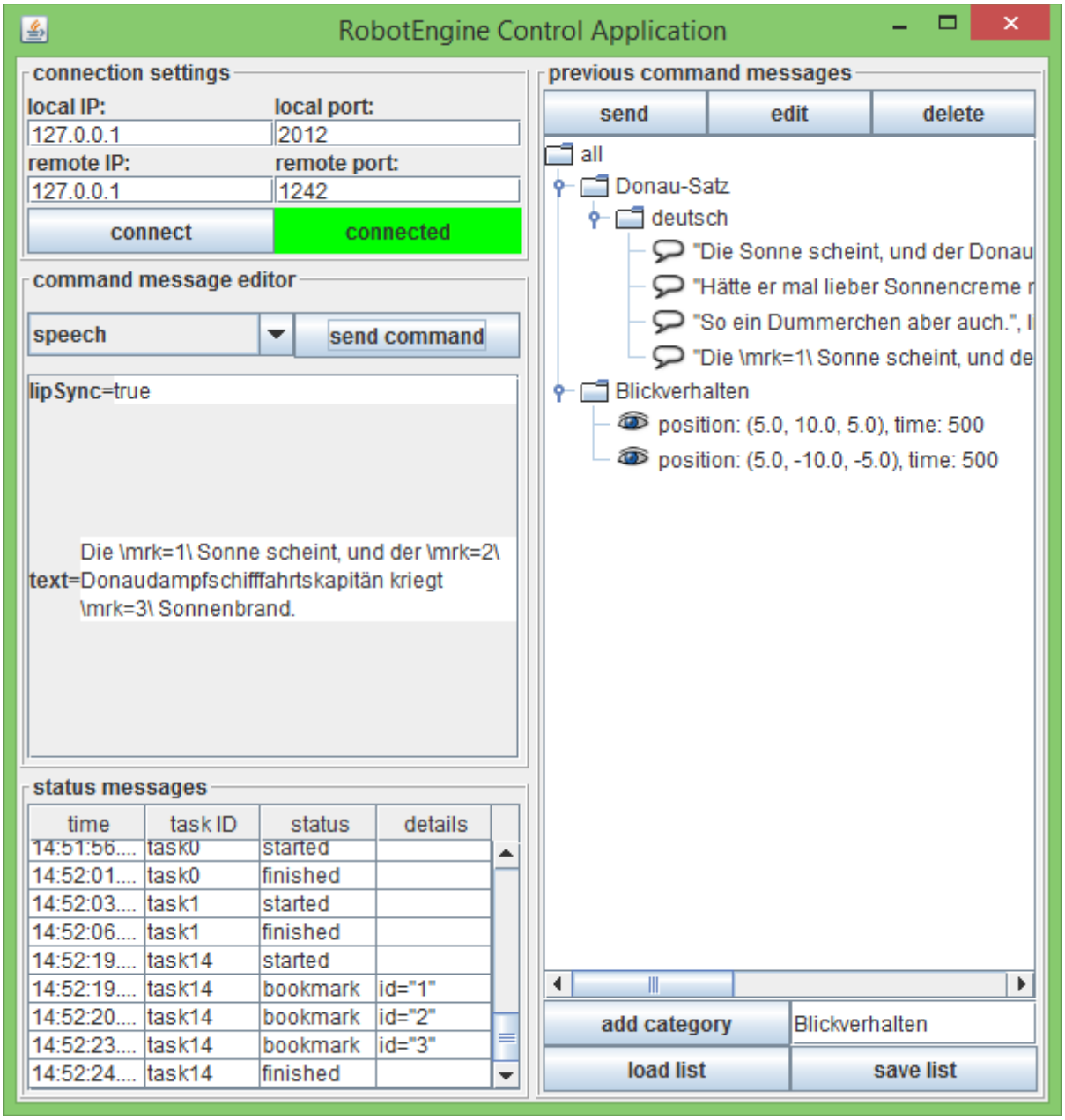


- kompatibel zu allen Kontroll-Anwendungen, welche das selbe Nachrichtenprotokoll verwenden
  - z.B. *de.hcm.robots.controlApp.DefaultControlApplication*
  - z.B. ein geeigneter Executor in Visual SceneMaker

- Variante B: direkte Einbettung in die Kontroll-Anwendung
  - mögliche Anwendungsfälle:
    - eigenständiges Wizard-of-Oz-Interface
    - Animations-Editor
  - Ausführung von Kommandos:
    - myRobotEngine.executeCommand(myCommandMessage);
  - Zugriff auf Status-Meldungen:
    - Kontroll-Anwendung muss das Interface “StatusMessageHandler” implementieren
    - Kontroll-Anwendung als Empfänger anmelden:
      myRobotEngine.setStatusMessageHandler(myControlApp);



Wie verwendet man die DefaultControlApplication?



- <u>Start und Verbindungsaufbau</u>
  - Hauptklasse *de.hcm.robots.controlApp.DefaultControlApplication,* keine weiteren Argumente nötig
  - IP-Adressen und Ports für die UDP-Client-Verbindung eintragen
  - auf "connect" klicken
- <u>Steuerung der *RobotEngine*</u>
  - gewünschten Aktions-Typ aus der Liste auswählen (links von "send command")
  - angezeigte Parameter nach Wunsch einstellen
    - Pflicht-Parameter sind mit * markiert
    - Sonderfall bei Typ "other...": *CommandMessage* direkt im XML-Format eingeben
  - auf "send command" klicken
  - empfangene Status-Meldungen werden in der unteren Tabelle aufgelistet

- Liste häufig benötigter Kommandos
  - Kategorien anlegen:
    - Namen in das Textfeld unter der Baumansicht eintragen
    - links neben diesem auf “add category” klicken
    - neue Kategorie mit angegebenem Namen wird innerhalb der Kategorie angelegt, die davor ausgewählt war
  - Kommandos hinzufügen:
    - geschieht automatisch bei Klick auf “send command”
    - gesendetes Kommando wird immer zur aktuell ausgewählten Kategorie hinzugefügt
  - ausgewähltes Element löschen: Klick auf “delete”
  - gespeichertes Kommando verwenden:
    - Klick auf “send” über der Baumansicht schickt das ausgewählte Kommando
    - Klick auf “edit” lädt eine Kopie des Kommandos in den Editor, wo die Parameter abgewandelt werden können
  - persistente Kommando-Listen:
    - “save list” öffnet einen Dialog, um die Liste als XML-Datei abzuspeichern
    - “load list” öffnet einen Dialog, um eine existierende Datei zu öffnen

Wie implementiert man eine RobotEngine?

1. Die Startmethode implementieren
   - zu überschreibende Methode: *public void start(String configPath)*
   - die Konfiguration laden
     - *loadConfig(configPath);*
     - Parameter auslesen, z.B:
       - *mEngineConfig.getProperty("voice", "Stefan");*
       - *mEngineConfig.getProperty("language", "de");*
       - *mEngineConfig.getProperty("volume", "100");*
   - die Verbindung zum Roboter und dessen einzelnen Services herstellen
   - Instanzen von Hilfsklassen und -threads anlegen

2. Die Stopmethode implementieren
   - zu überschreibende Methode: *public void stop()*
   - die Verbindung zum Roboter trennen
   - falls möglich, laufende Aktionen abbrechen
   - alle Hilfsthreads stoppen

3. Die Execute-Methode implementieren
   - zu überschreibende Methode: *public void executeCommand(CommandMessage cmd)*
   - erste Filterung nach Aktions-Typ:
     - *String type = cmd.getCommandType();*
     - *if(type.equals("speech")) …*
     - *else if(type.equals("anim") || type.equals("animation"))…*
     - *else: rejectCommand(cmd.getTaskID, "not supported");*
   - Auslesen der Aktions-Parameter:
     - *String text = cmd.getParam("text");*
     - *String fileName = cmd.getParam("name");*
     - *String color = cmd.getParam("color");*
     - *String lipSync = cmd.getParam("lipSync");*
     - ...
   - Zugriff auf Roboter-API, um die Aktion auszuführen

4. Die Ausführung der Aktion überwachen
   - Roboter-spezifisch, z.B.
     - Events abonnieren / abfangen
     - Rückgabewerte beim Methoden-Aufruf interpretieren
     - Variablen regelmäßig manuell prüfen
     - spezielle Scheduling-Lösungen, z.B. auf Basis von *de.hcm.robots.util.FIFOSpeechScheduler*
   - Fortschritte in Status-Meldung übersetzen:
     - Aktion abgeschlossen:
       - *StatusMessage status = new StatusMessage(taskID, "finished");*
       - *sendStatusMessage(status);*

    - Bookmark erreicht:
        - *StatusMessage status = new StatusMessage(task, "bookmark");*
        - *status.addDetail("id", ttsEvent.bookmarkID);*
        - *sendStatusMessage(status);*
- minimal benötigte Status-Meldungen:
    - “finished”: Aktion abgeschlossen
    - “rejected”: Aktion nicht unterstützt oder aktuell nicht möglich (z.B. ungültige Parameter)
    - “bookmark”: Sprachausgabe hat eine bestimmte Markierung erreicht
- optionale Status-Meldungen: z.B.
    - “started”: Ausführung der Aktion hat begonnen
    - “pending”: Kommando ist angekommen, aber wartet noch auf Ausführung (z.B. Ressourcenkonflikt mit bereits laufender Animation)
    - “word”: Sprachausgabe hat das nächste Wort erreicht
    - ...

5. Empfehlung: Wrapper-Klassen für zusammenhängende Funktionen

- bündeln alles, was mit einer bestimmten Aktions-Kategorie zu tun hat
    - Zugriff auf einen speziellen Service (z.B. Animation) oder verwandte Services (z.B. Text-To-Speech und MP3-Wiedergabe)
    - zugehörige Überwachungsmechanismen (z.B. TTS-Fortschritt oder Motorpositionen)
    - zugehörige Berechnungen (z.B. inverse Kinematik)
- Vorteile:
    - erhöht die Übersichtlichkeit
    - je nach Roboter-API notwendig für Nebenläufigkeit, z.B. um ein Animationskommando zu starten, während die Sprachausgabe läuft

Wie implementiert man eine Kontroll-Anwendung?

1. allgemeine Grundprinzipien
- Zuständigkeit
  - RobotEngine: Roboter-/Agenten-spezifische Aufgaben
    - Auflösung von Resourcenkonflikten
      - z.B. neues Sprachkommando zwischenspeichern, wenn das vorherige noch nicht abgeschlossen ist
      - z.B. vor Durchführung der Animation prüfen, ob die entsprechenden Motoren bereits in Verwendung sind
      - z.B. schonende Interpolation zwischen Motorpositionen
    - Übersetzung des Kommandos in Befehle der Roboter-/Agenten-API
    - Zugriff auf Hard- und Software des Roboters/Agenten
  - Kontroll-Anwendung: Interaktions- und höhere Verhaltenslogik
    - z.B. automatische Kopf-, Augen- und Blinzelbewegungen
    - z.B. Reaktion auf Nutzereingaben
    - z.B. Dialogablauf
    - z.B. Emotionsmodell
- nach Möglichkeit vorhandene Klassen nutzen
  - einheitliche Kommunikation sicherstellen über Paket *de.hcm.robots.messaging*
  - nützliche GUI-Komponenten befinden sich im Paket *de.hcm.robots.controlApp*

2. Verbindung zur *RobotEngine*
- das Interface *StatusMessageHandler* implementieren, um auf Status-Meldungen der *RobotEngine* zu reagieren
- Verbindung herstellen
  - Variante A: UDP-Verbindung
    - eine Instanz von *MessageClient* anlegen
      - zu verwendender *StatusMessageHandler*
      - Puffergröße
      - IP-Adresse und Port, auf dem die Kontroll-Anwendung Status-Meldungen empfängt
      - IP-Adresse und Port, auf dem die RobotEngine-Anwendung Kommandos empfängt
    - Client-Thread mit *start()* aktivieren
    - bei Bedarf Verbindung mit *abort()* trennen (z.B. beim Schließen der Anwendung oder vor erneutem Aufbau der Verbindung)
  - Variante B: direkter Zugriff
    - Instanz der gewünschten *RobotEngine* anlegen
    - verwendeten *StatusMessageHandler* mit *setStatusMessageHandler()* zuweisen

3. Senden von Kommandos

- *CommandMessage* anlegen
  - möglichst eindeutige Task-ID generieren
  - Kommando-Typ angeben
  - benötigte Parameter mit *addParameter(name, value)* zum hinzufügen
- an RobotEngine übermitteln
  - Variante A: über *MessageClient* an externe Instanz versenden
  - Variante B: über *executeCommand()* an lokale Instanz übergeben
- falls gewünscht, Task-ID speichern, um den Fortschritt zu überwachen

4. Verarbeiten von Status-Meldungen

- *MessageClient* oder interne *RobotEngine* ruft direkt *handleStatusMessage()* des zugeordneten *StatusMessageHandler* auf
- Task-ID der Status-Meldung ermöglicht Zuordnung zum Kommando, welches diese ausgelöst hat
  - z.B. Sprachausgabe, welche gerade die angegebene Markierung erreicht hat
  - z.B. Animation, welche gerade beendet wurde
- enthaltener Status: grundlegende Information
  - z.B. “finished” nach Ende der Durchführung
  - z.B. “rejected” bei gescheiterter Durchführung
  - z.B. “bookmark” beim Erreichen einer Markierung in der Sprachausgabe
- optionale Details:
  - z.B. “reason” bei gescheitertem Kommando klärt über die Ursache auf
  - z.B. “id” identifiziert die erreichte Markierung in der Sprachausgabe