**SQ Projekt 9 – Notizblatt**

Possible criteria for selecting model-based testing tools (relevant for methodology):

* Programming language (if specified for a tool): For TiGL C++ (mainly), C, CMake, Python, Java, SWIG relevant
* Tools for embedded systems, web or mobile applications not relevant for TiGL
* Also consider commercial tools? (if yes, then most likely no prototype) (possible question for Prof?)
* [More criteria could be added in terms of what exactly we’ll test from TiGL, e.g. GUI or specific feature (might possibly be specified with the dev contact)]

Initial list of possible model-based testing tools:

Sources:

<https://automated-360.com/model-based-testing/model-based-testing/>

<https://www.softwaretestingmagazine.com/tools/open-source-model-based-testing-tools/>

* **GraphWalker**:
  + addresses State Transition Model-Based Testing; in other words, it allows you to perform modeling around states and transitions between those states using directed graphs (Source: <https://docs.getxray.app/display/XRAY/Model-Based+Testing+using+GraphWalker+and+Java>)
  + seems it focuses on Java projects, integration as a maven dependency possible (Source & Download: <https://graphwalker.github.io/>)
  + Repo (incl. an example): <https://github.com/GraphWalker>
* **fMBT**:
  + suitable for testing anything from individual C++ classes to GUI applications and distributed systems (Source & Repo: <https://github.com/intel/fMBT>)
* **Modbat**:
  + specialized for API of software, compatible with Java (Source & Repo: <https://gitlab.com/cartho/modbat>)
* **Modelator:**
  + enables automatic generation of tests from models. Modelator takes TLA+ models as its input and generates tests that can be executed against an implementation of the model. (Source & Repo: <https://github.com/informalsystems/modelator>)
  + Requires Python or Java (more info in README in Repo)
* **OSMO MBT Tool:**
  + Tool for generating and executing test cases; expressed as Java programs (Source & Repo: <https://github.com/osmo-tool/osmo>)
* **Tcases:**
  + tool for designing tests. It doesn't matter what kind of system you are testing -- UI, command line, REST-ful API, or backend. Nor does it matter what level of the system you are testing -- unit, subsystem, or full system. You can use Tcases to design your tests in any of these situations. With Tcases, you define the input space for your system-under-test and the level of coverage that you want. Then Tcases generates a minimal set of test cases that meets your requirements. primarily a tool for black-box test design (Source & Repo: <https://github.com/Cornutum/tcases>)

Zusammenfassung Titanium Graphical Library:

TiGL, or the Titanium Graphical Library, is a software library primarily used for the parametric modeling and design of aircraft wings. It provides a set of tools and functions that allow engineers and designers to create, modify, and analyze complex wing geometries. TiGL is particularly useful in the aerospace industry for tasks such as aerodynamic analysis, structural design, and overall aircraft performance optimization. It's like a digital toolbox for crafting the wings of the future!  
  
Features TiGL:

TiGL comes with a range of features that make it a powerful tool for aircraft design.

1. **Parametric Modeling:** TiGL allows users to create and manipulate wing geometries using parameters. This means you can define the characteristics of a wing (like span, sweep, dihedral, etc.) through variables, making it easier to explore various design options.
2. **Geometry Generation:** It provides functions to generate complex 3D wing surfaces based on user-defined parameters. This is crucial for creating realistic and aerodynamically sound wing shapes.
3. **CAD Integration:** TiGL is often integrated with Computer-Aided Design (CAD) software. This enables seamless collaboration between different stages of the design process. Users can export or import wing geometries to and from CAD tools.
4. **Aerodynamic Analysis:** The software includes tools for aerodynamic analysis, helping engineers assess the performance of different wing configurations. This can involve studying lift, drag, and other aerodynamic forces to optimize the design for efficiency.
5. **Structural Analysis:** TiGL can assist in structural analysis, evaluating how well a wing design can withstand various loads and stresses. This is crucial for ensuring the safety and reliability of the aircraft.
6. **Mesh Generation:** Generating a mesh is a crucial step in numerical simulations. TiGL facilitates the creation of computational meshes for finite element analysis, which is used in structural and aerodynamic simulations.
7. **Data Exchange:** TiGL supports various data formats for exchanging information with other engineering tools. This interoperability is essential for a comprehensive design workflow.
8. **Visualization:** The software offers visualization tools to help users inspect and analyze wing geometries in a graphical interface. This aids in understanding the design and making informed decisions.

Example on how to test TiGL with Model-based testing:

Model-based testing (MBT) is a systematic testing approach where a model of the system under test is used to design and execute tests. In the case of TiGL, you could create a model that represents the expected behavior and specifications of the software, and then generate test cases based on that model.

Here's a simplified approach to model-based testing for TiGL:

1. **Define the Model:** Create a model that captures the different states and transitions within TiGL. This could include states like "Parameter Definition," "Geometry Generation," "Aerodynamic Analysis," etc. Define how the software should behave in each of these states and the possible transitions between them.
2. **Identify Test Scenarios:** Based on the model, identify various test scenarios that cover different aspects of TiGL's functionality. This could include testing different parameter combinations, assessing the software's response to extreme values, and ensuring proper integration with CAD software.
3. **Generate Test Cases:** Use the model to automatically generate test cases. These test cases should represent different paths through the model, covering a variety of inputs and conditions.
4. **Execute Tests:** Run the generated test cases on the actual TiGL software. Observe how the software behaves in response to the inputs and check if it adheres to the expected behavior defined in the model.
5. **Analyze Results:** Evaluate the results of the tests. Identify any deviations from the expected behavior and assess the robustness and reliability of the software under different conditions.
6. **Iterate and Refine:** If issues are found, iterate on the model and test cases, refining them based on the observed behavior. This iterative process helps improve the quality and coverage of the testing.

Sources:

<https://atos.net/wp-content/uploads/2017/05/atos-how-to-make-the-most-of-model-based-testing.pdf>

<http://www.model-based-testing.de/data/MBT_Hype_oder_Realitaet_OS_06_11.pdf>  
<http://www.model-based-testing.de/data/weissleder_phd_thesis.pdf>

Example:  
  
Specific example for model-based testing in the context of TiGL. Focus on the parameterization and geometry generation aspects.

**Model Definition:**

1. **State 1: Parameter Definition**
   * TiGL allows users to define parameters like wing span, sweep angle, and dihedral angle.
   * Transitions to the next state when parameters are successfully defined.
2. **State 2: Geometry Generation**
   * TiGL generates a 3D wing surface based on the defined parameters.
   * Transitions back to Parameter Definition state if there are invalid parameter values.

**Test Scenario:**

**Test Case 1: Valid Parameterization and Geometry Generation**

1. Set parameters: Span = 30 meters, Sweep = 25 degrees, Dihedral = 5 degrees.
2. Verify that the software successfully transitions from Parameter Definition to Geometry Generation.
3. Check if a 3D wing surface is generated without errors.
4. Ensure the generated wing shape adheres to the specified parameters.

**Test Case 2: Invalid Parameterization Handling**

1. Set parameters: Span = -10 meters (invalid value), Sweep = 20 degrees, Dihedral = 0 degrees.
2. Verify that TiGL detects the invalid parameter (negative span) and remains in the Parameter Definition state.
3. Confirm that appropriate error messages or warnings are displayed.

**Test Case 3: Robustness Testing**

1. Set parameters: Span = 40 meters, Sweep = 30 degrees, Dihedral = 10 degrees.
2. Introduce unexpected interruptions during geometry generation (e.g., simulate a sudden system interruption or resource shortage).
3. Verify that TiGL gracefully handles interruptions, preserving the integrity of the model and providing appropriate recovery mechanisms.

These test cases cover different aspects of the parameterization and geometry generation process, ensuring that TiGL behaves as expected under various conditions. The model guides the creation of these scenarios and helps ensure comprehensive testing of the software.

Questions to Prof. Felderer:   
  
-> We want to present the TiGL software and its features, then give a broad overview about model based testing, and related tools and models.

-> Afterwards we want to explain which model we selected and how applied to TiGL, results.   
-> What should the Methodology contain?   
-> 1) Intro 2) Background 3) Methodology 4) Results 5) Discussion

-> Which models for model-based testing would you recommend for TiGL?

->   
  
Meeting mit Prof Felderer:  
  
Werkzeug auswählen, wie ausgewählt, warum tigl, welche Probleme?     
  
Prozess wie man vorgegangen ist.     
- abstraktes Testmodell erstellen, Umsetzung.    
TiGL viewer testen, GUI. Testing.     
  
Zustandsmaschine ->   Datenstrukturen -> Files eingeben -> und schauen ob das gleiche wieder herauskommt.     
  
Mathematische Modellierung zu testen? Könnte schwer werden.     
  
TiGL Vorstellung nur kurz, weil ist nur das Programm nur ein Beispiel ist.

Questions to the developer of TiGL:

1. **Understanding TiGL's Behavior:**
   * Can you provide examples of common user scenarios or workflows in TiGL?
   * Are there any specific user inputs or actions that significantly impact the behavior of TiGL?
2. **Handling Edge Cases:**
   * Can you share insights into how TiGL handles edge cases or unexpected inputs from users?
   * Are there specific error messages or warnings that users should be aware of?
3. **Dependencies and External Factors:**
   * Are there any external dependencies or factors that can influence TiGL's behavior (e.g., system configurations, third-party libraries)?
   * How robust is TiGL in handling variations in the environment?
4. **User Feedback and Pain Points:**
   * Based on user feedback, are there any common pain points or challenges that users face when working with TiGL?
   * Have there been notable issues or areas of improvement identified by users in the past?
5. **Performance Considerations:**
   * Are there performance considerations or limitations that testers should be aware of, especially when dealing with large or complex aircraft designs?
   * How does TiGL manage resource utilization during intensive computations?
6. **Model Extensibility:**
   * Can you provide insights into the extensibility of TiGL? How easy is it for users to customize or extend functionality for specific needs?
   * Are there any guidelines or best practices for users who want to enhance TiGL for their specific applications?
7. **Documentation and User Support:**
   * How comprehensive is the documentation for TiGL, and are there specific sections that testers should pay close attention to?
   * What avenues of user support are available, such as forums, user communities, or direct support from the TiGL team?
8. **Real-world Use Cases:**
   * Can you share examples of real-world projects or applications where TiGL has been successfully used?
   * Are there any notable success stories or challenges that users have overcome with TiGL?
9. **Collaboration with Developers:**
   * How does the TiGL development team collaborate with users and testers for feedback and improvement?
   * Are there any beta testing programs or opportunities for users to contribute to the development process?

Meeting with jan:   
  
Unit tests in die gui ziehen und schauen ob es funktioniert.  
Screenshot vom File und vorher nachher abgleichen.  
Examples ordner: Python scripte mit Anwendungsbeispielen (also Tests).  
Python wird nicht getestet.  
CPACS File - > CUT File   
Strukturen einfügen etc.

Drei Beispiele für Test Cases GUI:

**Testfall 1: Öffnen und Anzeigen eines Modells**

**Voraussetzungen:**

* Der TiGL Viewer ist installiert.
* Es gibt ein gültiges Modell, das getestet werden kann.

**Testschritte:**

1. Starte den TiGL Viewer.
2. Überprüfe, ob die Anwendung ordnungsgemäß geöffnet wird.
3. Wähle im Menü die Option "Datei öffnen" aus.
4. Navigiere zu einem gültigen Modell und wähle es aus.
5. Überprüfe, ob das Modell erfolgreich geladen wird.
6. Navigiere durch die verschiedenen Ansichten des Modells.
7. Überprüfe, ob die Ansichten korrekt angezeigt werden.
8. Schließe den TiGL Viewer.

**Erwartete Ergebnisse:**

* Die Anwendung öffnet sich ohne Fehler.
* Das ausgewählte Modell wird erfolgreich geladen.
* Alle Ansichten des Modells werden korrekt dargestellt.
* Die Anwendung schließt sich ordnungsgemäß.

**Testfall 2: Komponentenauswahl und Rotation**

**Voraussetzungen:**

* Der TiGL Viewer ist installiert.
* Es gibt ein gültiges Modell mit mehreren Komponenten.

**Testschritte:**

1. Starte den TiGL Viewer und öffne ein Modell.
2. Wähle eine Komponente im Modell aus.
3. Überprüfe, ob die ausgewählte Komponente hervorgehoben wird.
4. Rotiere die ausgewählte Komponente um einen bestimmten Winkel.
5. Überprüfe, ob die Rotation korrekt auf die Ansicht angewendet wird.
6. Wähle eine andere Komponente aus.
7. Überprüfe, ob die vorherige Auswahl aufgehoben wird.
8. Schließe den TiGL Viewer.

**Erwartete Ergebnisse:**

* Die ausgewählte Komponente wird hervorgehoben.
* Die Rotation der ausgewählten Komponente wird korrekt angezeigt.
* Die Auswahl wird aufgehoben, wenn eine andere Komponente ausgewählt wird.
* Die Anwendung schließt sich ordnungsgemäß.

**Testfall 3: Laden eines Modells mit ungültiger Datei**

**Voraussetzungen:**

* Der TiGL Viewer ist installiert.

**Testschritte:**

1. Starte den TiGL Viewer.
2. Wähle im Menü die Option "Datei öffnen" aus.
3. Navigiere zu einer Datei, die kein gültiges Modell enthält (z.B., eine nicht unterstützte Dateierweiterung oder eine beschädigte Datei).
4. Wähle die ungültige Datei aus.
5. Überprüfe, ob eine Fehlermeldung angezeigt wird und das Modell nicht geladen wird.
6. Schließe die Fehlermeldung.
7. Überprüfe, ob die Anwendung in einem stabilen Zustand bleibt.

**Erwartete Ergebnisse:**

* Eine Fehlermeldung wird angezeigt, die darauf hinweist, dass die Datei nicht geladen werden kann.
* Das Modell wird nicht geladen.
* Die Anwendung bleibt in einem stabilen Zustand, auch nach dem Versuch, eine ungültige Datei zu öffnen.

Dieser Testfall zielt darauf ab, die Anwendung auf ihre Reaktion bei fehlerhaften Dateien zu überprüfen und sicherzustellen, dass sie in einem stabilen Zustand bleibt, selbst wenn das Laden eines ungültigen Modells fehlschlägt.

Die Testfälle decken grundlegende Interaktionen mit dem TiGL Viewer ab, darunter das Öffnen von Modellen, das Navigieren durch Ansichten und das Testen von Auswahl- und Rotationsfunktionen.

**Begründung für das GUI Testing:**

Das GUI-Testing des TiGL Viewers ist von entscheidender Bedeutung, selbst wenn in der früheren Entwicklungsphase das GUI nicht ausgiebig getestet wurde. Hier sind einige Gründe, warum das GUI-Testing wichtig ist:

1. **Anwendererfahrung verbessern:**
   * Das GUI ist die Benutzerschnittstelle der Anwendung. Durch das Testen des GUIs können potenzielle Benutzererfahrungen, Benutzerfreundlichkeit und das allgemeine Erscheinungsbild verbessert werden.
2. **Fehlererkennung und -behebung:**
   * GUI-Tests helfen dabei, mögliche Fehler oder Inkonsistenzen in der Benutzeroberfläche zu identifizieren. Auch wenn das GUI während der Entwicklung nicht intensiv getestet wurde, bietet GUI-Testing die Möglichkeit, eventuelle Probleme aufzudecken und zu beheben.
3. **Stabilität sicherstellen:**
   * GUI-Tests tragen dazu bei, die Stabilität der Anwendung sicherzustellen, insbesondere wenn neue Funktionen hinzugefügt oder Änderungen am Code vorgenommen werden. Sie helfen dabei, sicherzustellen, dass GUI-Änderungen keine negativen Auswirkungen auf die Anwendungsstabilität haben.
4. **Kundenzufriedenheit gewährleisten:**
   * Ein stabiles und gut funktionierendes GUI trägt zur Zufriedenheit der Endbenutzer bei. Das Testen des GUIs ermöglicht es, sicherzustellen, dass die Anwendung den Erwartungen der Benutzer entspricht.
5. **Risiken minimieren:**
   * Ohne GUI-Tests besteht das Risiko, dass Benutzer auf unerwartete Probleme stoßen. Durch das systematische Testen des GUIs können potenzielle Risiken minimiert und die Gesamtqualität der Anwendung verbessert werden.
6. **Nachhaltige Entwicklung fördern:**
   * Durch das Einbinden von GUI-Tests in den Entwicklungsprozess wird eine nachhaltige Entwicklung gefördert. Es ermöglicht Entwicklern, frühzeitig auf Probleme zu reagieren und sicherzustellen, dass GUI-Änderungen keine negativen Auswirkungen haben.

Obwohl das GUI möglicherweise nicht während der gesamten Entwicklung getestet wurde, bietet das nachträgliche Hinzufügen von GUI-Tests eine Möglichkeit zur Qualitätsverbesserung und zur Gewährleistung einer reibungslosen Benutzererfahrung. Es stellt sicher, dass die Anwendung nicht nur funktional, sondern auch visuell und benutzerfreundlich robust ist.