Exploring the Presence of Technical Debt in Industrial GUI-based Testware: A Case Study

Emil Alégroth, Marcello Steiner, Antonio Martini

2016-04-11
What is Technical Debt?

- Technical debt (TD) is a concept that describes the increased cost of development and maintenance of a system given that it is a sub-optimal solution.

- TD implies that software can be developed in an optimal way, e.g. optimized for:
  - Maintainability
  - Reusability
  - Etc.
Software vs Testware

• Software is designed and developed using structured development practices

• Testware is regarded as “only scripts”
  • Less structured development practices
  • Less verification of correctness
  • Less followed best practices

• Is this a good, or even viable, practice?
Methodology

- Exploratory case study at CompanyX where one member of the research team worked on location for 6 months.

- The study aimed at answering the research questions:
  - **RQ1**: What items associated with technical debt of software can be observed in industrial grade GUI-based testware?
  - **RQ2**: What technical debt items can be observed in practice that are unique to GUI-based testware?
Automated GUI-based testing

Pictorial GUI (on screen)
- GUI model
- GUI library
- GUI architecture
- API
- Etc.

System

Second (2nd) Generation
(Component-, tag-, widget-based)
- Tools: Selenium, QTP, RTteser, etc.
- Verification: Verifies that the system conforms to its requirements but not that the pictorial GUI conforms to the GUI model.

Third (3rd) Generation
Visual GUI Testing
- Tools: Sikuli, JAutomate, EggPlant, UFT, etc.
- Verification: Verifies that the system conforms to its requirements through input and assertions made to the GUI as shown on the screen.
Context

- Company with 3000 employees
  - 300 at studied location

- Safety critical software
  - Developed with agile development practices
  - Self-organizing teams
  - Each system in the range of 100k LOC

- Rigorous verification and validation
  - Low level: Thousands of Unit tests
  - Mid level: Hundreds of integration tests
  - High level: Hundreds of GUI tests with Unified functional testing (UFT) and manual testing
Case study

Contextual Analysis
- Document analysis, informal and semi-structured interviews

Data mining
- Semi-automated data mining of forums, issue-tracker and repositories

Analysis and Synthesis
- Thematic analysis with coding

Verification
- Semi-structured interviews
Data mining

- **Projects A-D**: Interviews and document analysis

- **Forum**: Qualitative information acquired through structured search strings
  - Test maintenance: 8467 entries
  - “Test maintenance”: 28 entries

- **Issue tracker**: Lacked structured search
  - Scripts extracted information to spreadsheets
  - Qualitative data analyzed formally

- **Analysis**:
  - Coding (Thematic analysis)
  - Cyclomatic complexity
  - Statement complexity
  - Single responsibility violations
RQ1: What items associated with technical debt of software can be observed in industrial grade GUI-based testware?

- **Function Complexity**: Functions that are unnecessarily complex, lower readability, etc. (Cyclomatic complexity)
- **DRY (Don’t repeat yourself) violations**: DRY violations in each repository, in each project, between projects.
- **God functions**: Methods that test different aspects of the system under test in the same test script.
- **Complex statements**: Long statements prohibit readability.
- **High arity**: A high number of input parameters and method calls caused by excessive modularization
RQ2: What technical debt items can be observed in practice that are unique to GUI-based testware?

• **Use of wrong UI-testing technology:**
  • Different benefits with different technologies
  • Often caused by developer preference
  • Lack of guidelines for structured/best suitable use

• **Use of monolithic object repositories**
  • Binary repositories of GUI representations
  • Stifles concurrent work since the repositories cannot be merged
Implications

- **TD can be found in testware!**
  - Testware requires equally stringent practices as software

- **TD can be automatically identified in testware!**
  - For instance using Cyclomatic complexity
  - However, the metric needs to be updated (Find suitable threshold)

- **There is best practice for developing testware!**
  - Testware requires equally stringent practices to software

- **The study only Identified a small set of TD items!**
  - More TD items common to software
  - More TD items unique to testware

- **Trade-off between testware modularization and readability**
  - High modularization: low readability, high reusability
  - Low modularization: High readability, low reusability
Conclusions

What is Technical Debt?

- Technical debt (TD) is a concept that describes the increased cost of development and maintenance of a system given that it is a sub-optimal solution.
- TD implies that software can be developed in an optimal way, e.g., optimized for:
  - Maintainability
  - Reusability
  - Etc.

Automated GUI-based testing

- Pictorial GUI (on screen)
- GUI model
- GUI library
- GUI architecture
- API
- Etc.
- System

- Third (3rd) Generation Visual GUI Testing
  - Tools: Sikuli, JAutomate, EggPlant, UFT, etc.
  - Verification:
    - Verifies that the system conforms to its requirements through input and assertions made to the GUI as shown on the screen.

Case study

- Contextual Analysis
  - Document analysis, informal, and semi-structured interviews
- Data mining
  - Semi-automated data mining of forums, issue-tracker, and repositories
- Analysis and Synthesis
  - Thematic analysis with coding
- Verification
  - Semi-structured interviews

Implications

- TD can be found in testware!
  - Testware requires equally stringent practices as software
- TD can be automatically identified in testware!
  - For instance using Code::Mate complexity
  - However, the metric needs to be updated (Find suitable threshold)
- There is best practice for developing testware!
  - Testware requires equally stringent practices to software
- The study only identified a small set of TD items!
  - More TD terms common to software
  - More TD terms unique to testware
- Trade-off between testware modularity and readability
  - High modularity: low readability, high reusability
  - Low modularity: high readability, low reusability
Questions?
Thank you for listening!
Results

- Legacy system
- Redevelopment of Legacy system
- Flight crew management
- Common, reusable, repository

(a) The measured cyclomatic complexity in Project A
(b) The measured cyclomatic complexity in Project B
(c) The measured cyclomatic complexity in Project C
(d) The measured cyclomatic complexity in Project D