HSLU Hochschule Luzern

Software Architecture and Techniques

Verify Functional Features

Marcel Baumann, tangly llc

Characteristics

- Change should be cheap
- You should have a feedback look, software design and development are an **empirical** activity
- Do not use speculation to add extra complexity
- Always think three things that might go **wrong**
- Work in **smaller teams** to produce **good software**

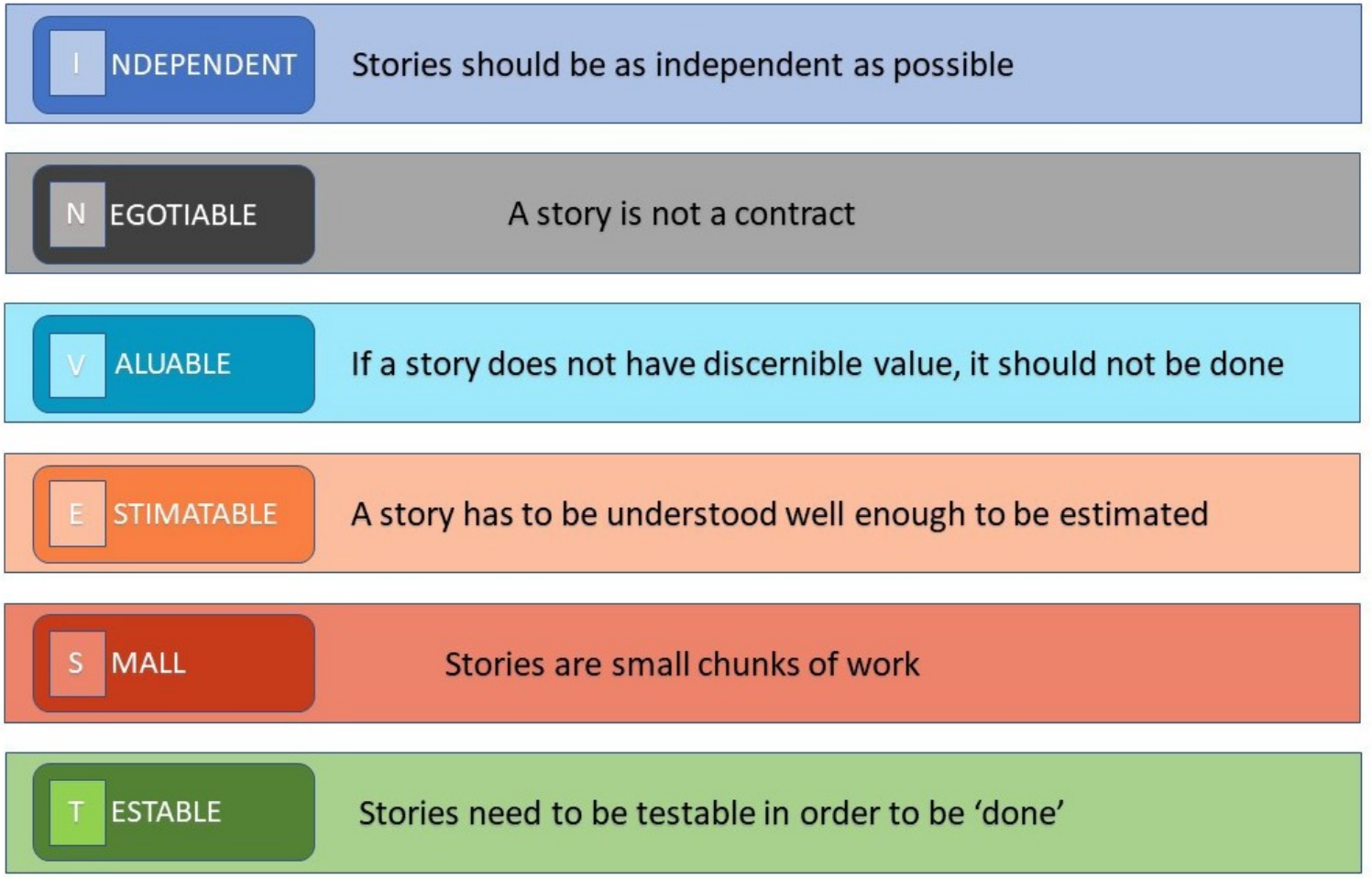
Agile Architecture Rules

- Features should be validated through tests
- Tests should be automated
- Tests should be run before each release to avoid regression errors
- Releases are performed multiple times per Sprint

Functional Requirements (1/2)

- **S** – Specific
- **M** – Measurable → **acceptance criteria**
- **A** – Attainable
- **R** – Realizable → within a sprint
- **T** – Traceable → **acceptance tests**

Stories as Functional Requirements (2/2)



Stories

- *As* [role] *I can* [function] *so that* [rationale]
- *As* a student, *I can* find my grades online *so that* I don’t have to wait until the next day to know whether I passed.
- Acceptance Criteria → Specification by Example
- A story should be **told** and trigger a **discussion**

Scrum and Stories

- A Scrum team always has a *Definition of Done*. All criteria of the *DoD* must be fullfilled to complete a story.
- A story has always acceptance criteria. All acceptance criteria shall be fullfilled to complete a story.
- Acceptance criteria shall be validated automatically to allow continuous integration and delivery.

Use Cases

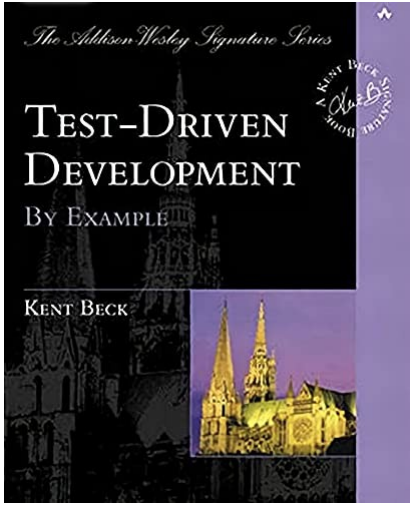
- Use Cases are dead. Just forget them.
  - Related Use Cases → Epics (*and use story maps*)
  - Primary Actors → Personas
  - Main Scenario → Story
  - Flow in Scenario → Discussion e.g. through refinement or event storming
  - Alternative Scenarios → Acceptance Criteria

Validation

- TDD
  - Safety net for refactoring and documentation by example
- ATDD
  - Subsystem level
  - System level – Java Modules or ArchUnit for some architecture validation –
- User Interface Tests
  - Selenium – *try to minimize their number -, they are brittle*

Test Driven Development *TDD*

- Validate the behavior of a class or a package
- Security net empowering you to refactor
- Should be part of definition of done
  DoD in Scrum



FIRST Unit Tests

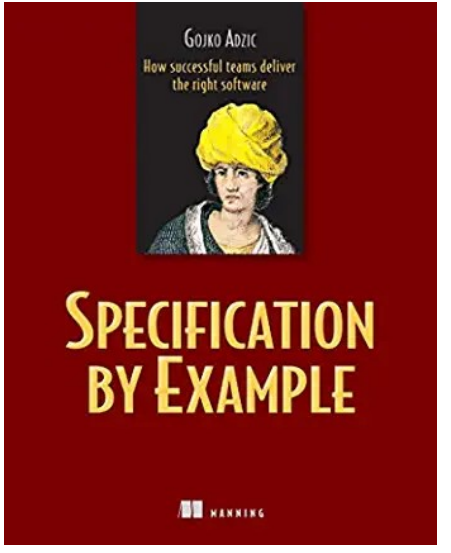
- Fast: *Many hundreds per seconds*
- Independent: *Failure reasons become obvious*
- Repeatable: *Run repeatably in any order*
- Self-validation: *No manual evaluation required*
- Timely: *Written before / during code*

TDD Tools

- JUnit 5
- AssertJ
- Mockito
- Always part of your CI/CD pipeline

Acceptance Test Driven Development *ATDD*

- Part of any story are acceptance criteria.
  - Acceptance criteria should be implemented as automated tests
  - All acceptance criteria should be executed before a release to mitigate regression issues
- Part of specification by example approach



ATTD Tools

- Same as with TDD: JUnit 5, AssertJ, Mockito
- Cucumber, Jbehave: *tools are stagnating*
  - Their technique example mapping is very similar to event storming in DDD
- Own libraries and approaches

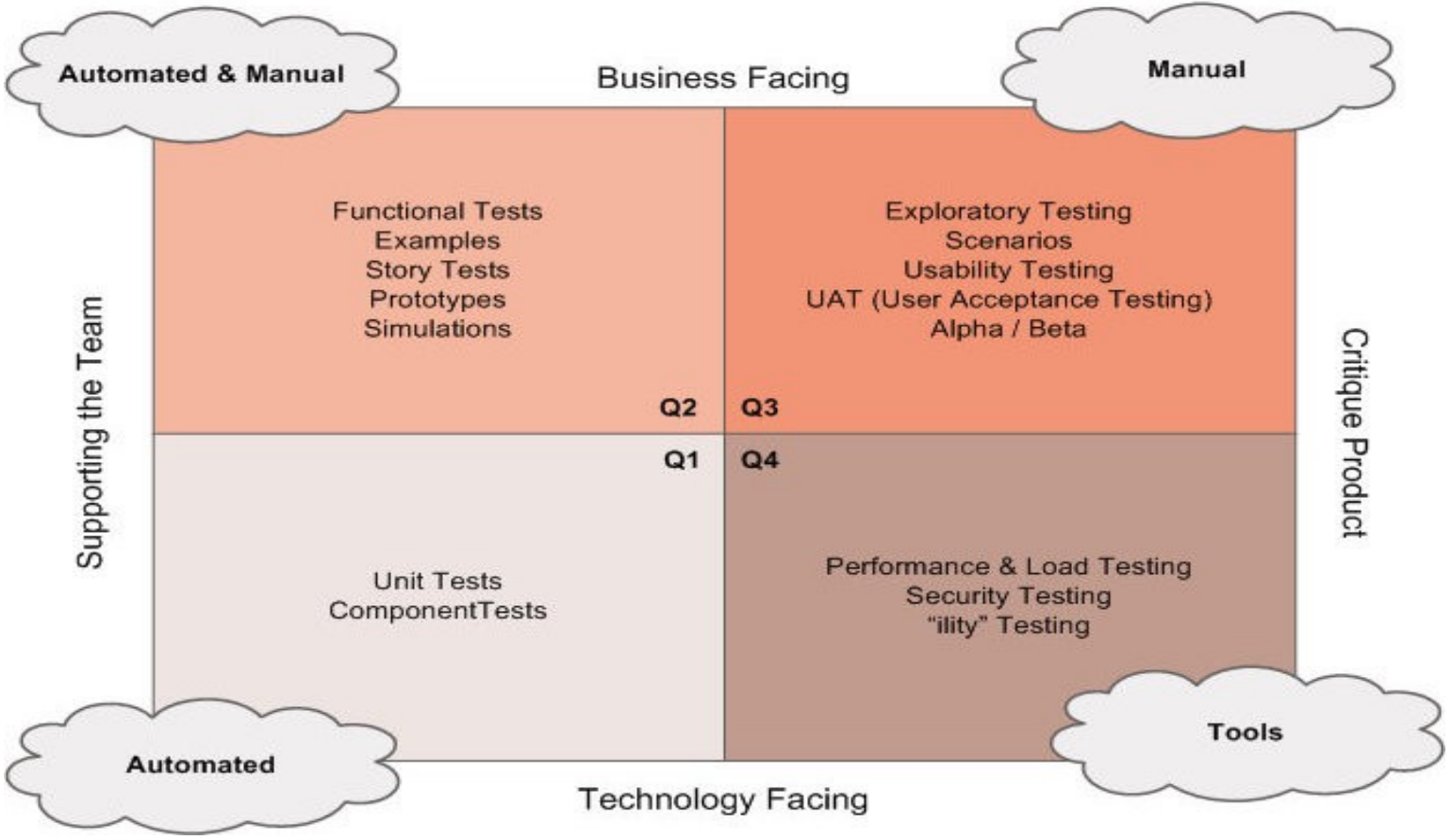
Interface Tests

- Interface are often either user interface or some REST services
- REST services define a contract with users and shall be tested as acceptance tests
- User interface are the window to your application

Interface Test Tools

- Services
  - OpenAPI, Swagger, Postman, Jmeter
- User Interface
  - Selenium

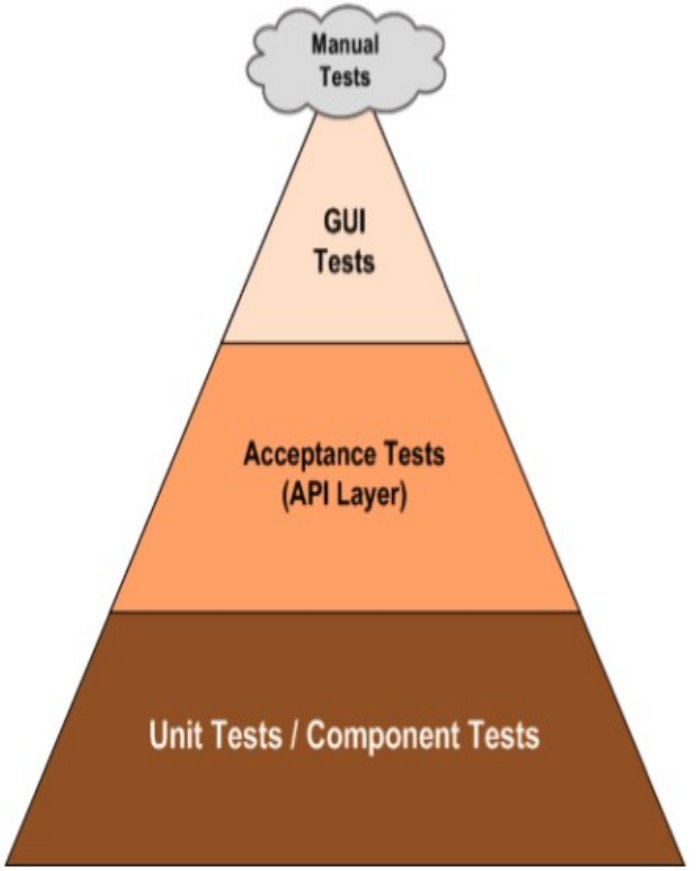
Testing Quadrants



Testing Pyramid

Automate all your tests:

- 4000 Unit Tests, 800 Acceptance Tests, 150 GUI Tests, 30 Manual Tests, 1 week “-ility” tests with 12 scenarios
- 2 weeks iteration, 1 year duration => 26 tests campaigns for a potentially shippable product
- 4 releases => 4 test campaigns for deployed product
- *Code is refactored in each sprint, every two weeks*



3 Verification Report

3.1 Summary

Number of test cases	passed	25
	failed	0
Total number of test cases	performed	25

3.2 List of Test Results

TC ID	TC Name	Author	Reviewer	Date / Time	Result
UTC291	RunDaily/WeeklyMainten	Peter Rey / pr	n/a	4/24/2009 10:31:58	PASSED
UTC292	AddInstr			:59	PASSED
UTC293	Connect			:01	PASSED
UTC294	Disconn Phoenix			:02	PASSED
UTC295	Impleme			:02	PASSED
UTC296	Instrume NotifyIns			:02	PASSED
UTC297	Instrume			:02	PASSED
UTC298	Instrume Mainten				SED
UTC299	Instrume				SED
UTC300	LogExce				SED
UTC301	LogMeth				SED

5.8 UTC298 - InstrumentInitializationMaintenanceRequired

ID	UTC298
Name	InstrumentInitializationMaintenanceRequired
Author	Peter Rey / pr
Reviewer	n/a
Description	If the ML_STAR instrument is switched on, the initialization of the ML_STAR instrument and the heater shaker was successful but there is outstanding maintenance, the instrument view shall be notified with the instrument status maintenance required
Test Methods	- Normal Case
Execution Date Time	4/24/2009 10:
Host ID	OLOS
User	peterrey
Environment	NUnit with Tes
Pre-Condition	None
Details	Description: SPI co Expected Outcome Outcome: Object is **PASSED**

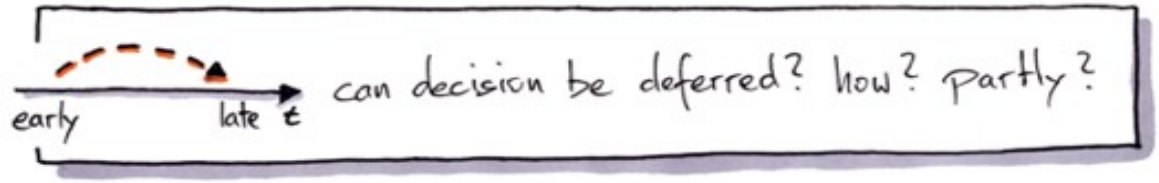
USP742
Criticality: Low
UTC298 InstrumentInitializationMaintenanceRequired
USP743
Criticality: High
UTC310 UnexpectedErrorOnInstrument
USP744
Criticality: Low

Architecture Goals

- **Reduce** Complexity
- **Increase** Changeability
- **Enable** Parallel Development

You have three programming paradigms: structured, object-oriented, and functional

Architecture Questions



- persist data of your system to survive restart
- how to translate UI and data
- communication between parts of your system
- scaling (run on multiple threads, processes, machines)
- security (how to authenticate, authorize)
- journaling (Activities, data)
- reporting
- data migration / data import
- releasability
- backwards compatibility
- response times
- Archiving data

design to be independent on decision

Quality Attributes

- Loose Coupling
- High Cohesion
- Design for Change
- Separation of Concerns
- Information Hiding
- *Good Practices: DDD, legibility of artifacts, git for traceability, infrastructure as code*

Quality Attributes

- Abstraction
- Modularity
- Traceability
- Decrease operating costs – *tracing, logging, monitoring -*
- Self documenting – *clean code* – **and** JavaDoc
- Incremental design

How Can You Reach These Goals?

- Spikes
- Experience and ask experts
- Codified knowledge – e.g. Java API, slf4j -
- Copy, modify, mutate, improve
- Refactor
- Unlock collective wisdom – *ask questions in forums! -*

Quality Citations

Lowering quality lengthens development time.
- First Law Of Programming

The quality of Code is inversely proportional to the effort it takes to understand it.

When I wrote this, only God and I understood what I was doing. Now God only knows.

Prefer good code over clever code.

Those who sacrifice quality to get performance may end up getting neither.

Reflection

- How can you learn faster?
- What should you change in your team to improve?
- How can you deliver better products?
- How can you improve quality of your products?

Links

- How to Build Quality Software Fast?, Dave Farley, GOTO 2022

Exercises (1/2)

- Unit Testing
- Module Testing
- Integration Testing
- Story Map Testing

Exercises (2/2)

- Read the optional architecture document
- Coding dojos
  - Implement and refactor a pattern – e.g. Builder, Factory Method, Factory -
  - Show your logging approach and associated code