**Table of Contents**

[Chapter 1: Brief History of Software Requirements Methods](#Chapter1) 2

[Intro to Software Requirements 7](#IntroToSoftwareRequirements)

[Chapter 2: The Big Picture of Agile Requirements 8](#Chapter2)

[Chapter 3: Agile Requirements for the Team 11](#Chapter3)

[Chapter 12: Requirements Discovery Toolkit 13](#Chapter12)

[Chapter 13: Vision, Features, Roadmap 16](#Chapter13)

[Requirements Elicitation 17](#RequirementsElicitation)

[Chapter 6: User Stories 19](#Chapter6)

[Chapter 7: Stakeholders, User Personas, and User Experiences 23](#Chapter7)

[Chapter 19: Use Cases 27](#Chapter19)

[Chapter 9: Activity Diagrams 29](#Chapter9)

[Use Cases and Scenarios 30](#UseCasesAndScenarios)

[Chapter 4: Agile Requirements for the Program 31](#Chapter4)

[Chapter 5: Agile Requirements for the Portfolio 34](#Chapter5)

[Chapter 17: Nonfunctional Requirements 37](#Chapter17)

[In-Class Presentations](#InClassPresentations)

**[Chapter 1: A Brief History of Software Requirements Methods Sept. 23](#TableOfContents)**

Migration towards more rapidly exploratory and lighter-weight processes

1970s 🡪 Predictive Processes (Waterfall)

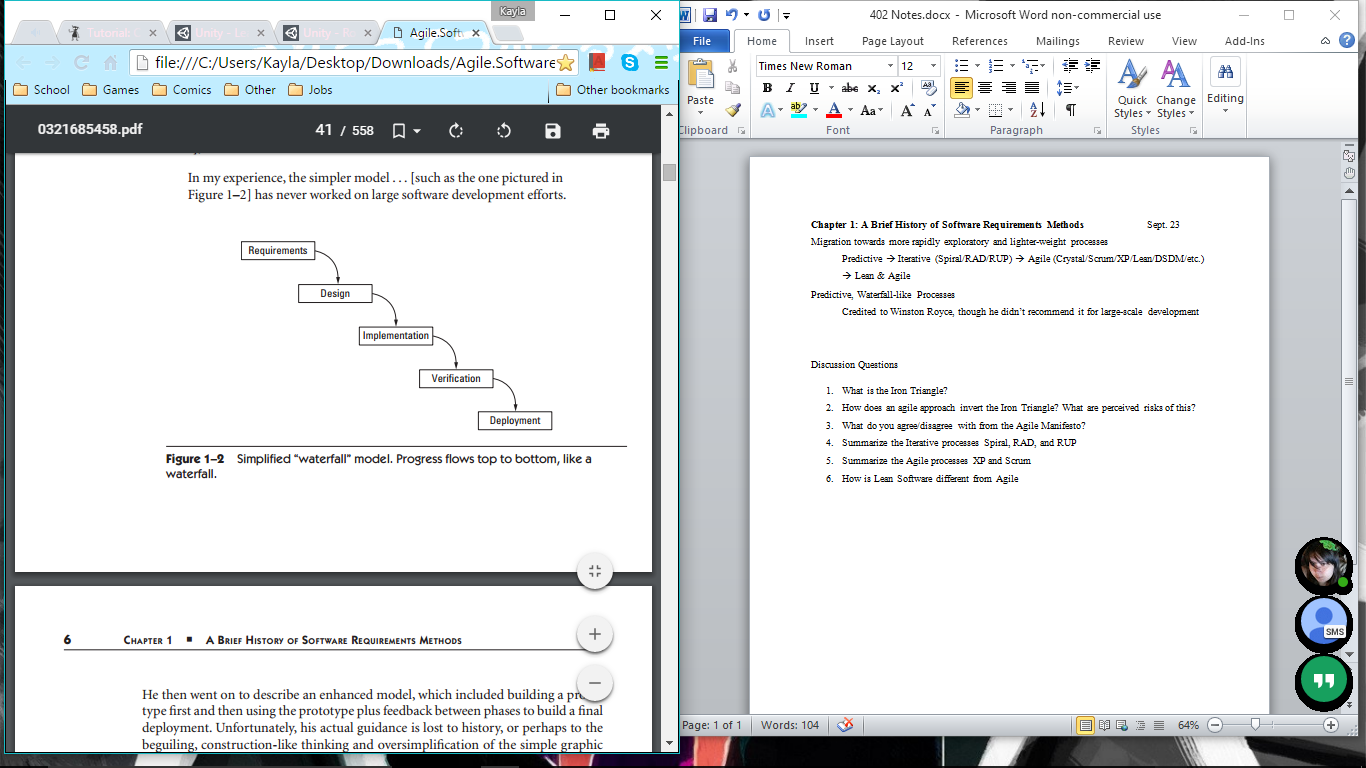
1980s 🡪 Iterative Processes (Spiral/RAD/RUP)

1990s 🡪 Adaptive (Agile) Processes (Crystal/Scrum/XP/FDD/Lean/DSDM/etc.)

2000s 🡪 Enterprise-Scale Adaptive (Lean & Agile) Processes

**Predictive, Waterfall-like Processes (1970s)**

Software development occurred in an orderly series of sequential stages

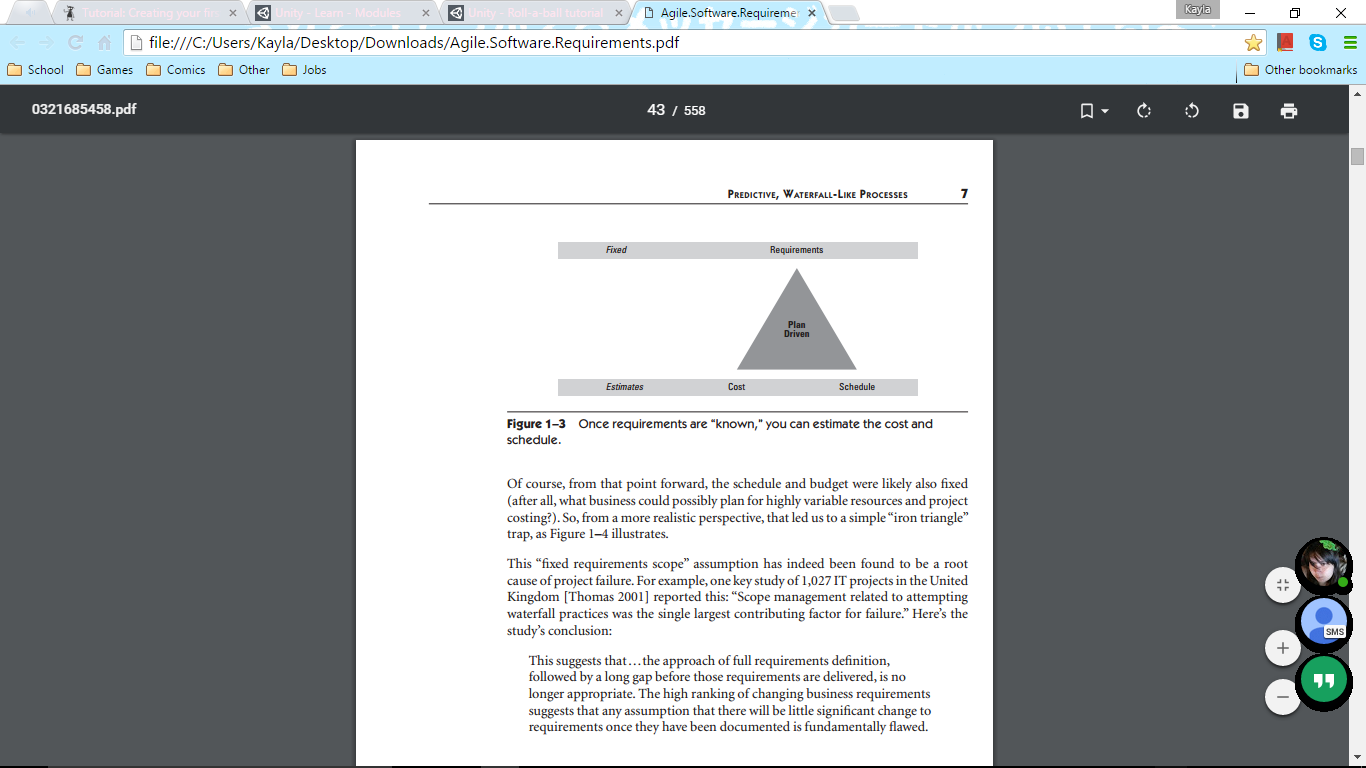


Credited to Winston Royce, though he didn’t recommend it for large-scale development

Problems: doesn’t work well for large software projects

Failures often caused by lack of user input, incomplete or changing requirements

Requirements: The Iron Triangle



Waterfall method assumes requirements can be determined up-front; not true

Requirements used as a basis to estimate cost/schedule, but often change

Causes cost and schedule to be fixed too, since they rely on requirements

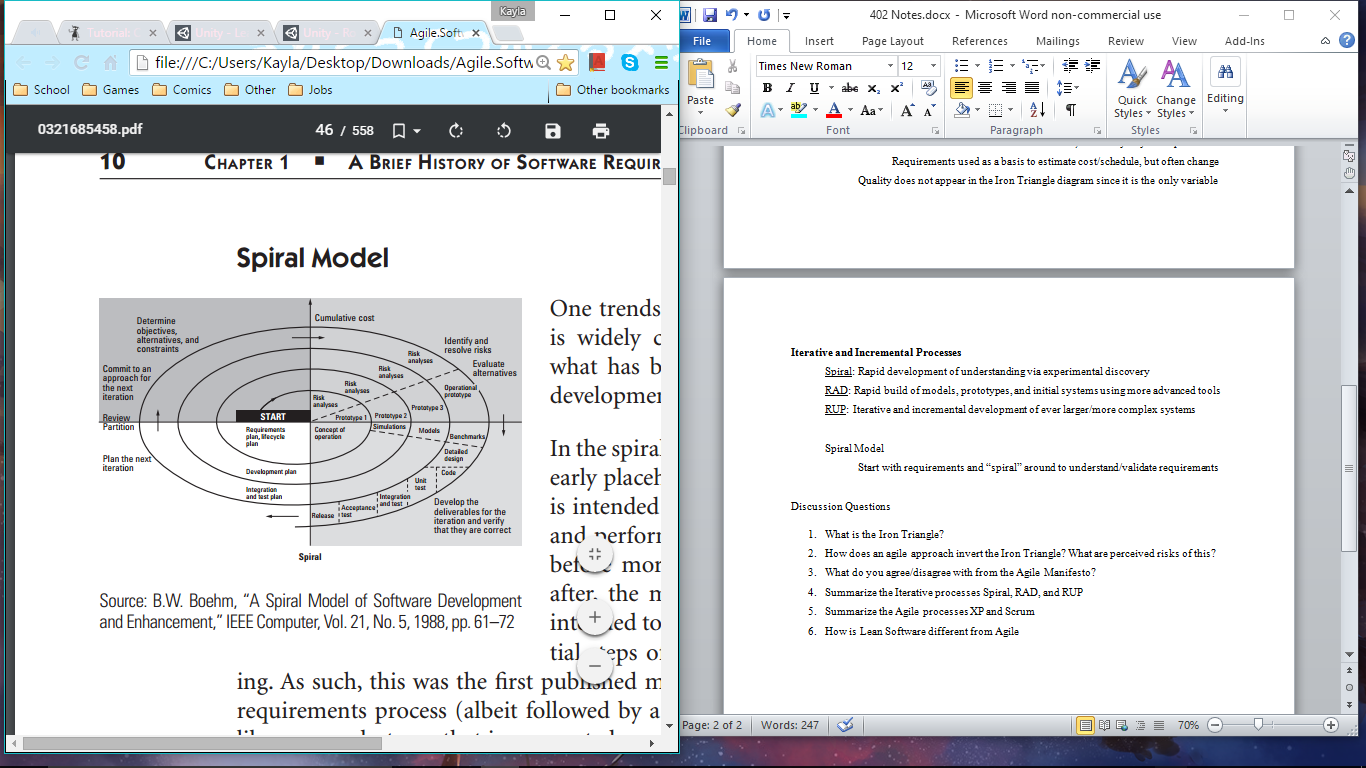
Quality does not appear in the Iron Triangle diagram since it is the only variable

**Iterative and Incremental Processes (late 80’s – early 90’s)**

Spiral: Rapid development of understanding via experimental discovery

Start with requirements, spiral around once to test/validate them

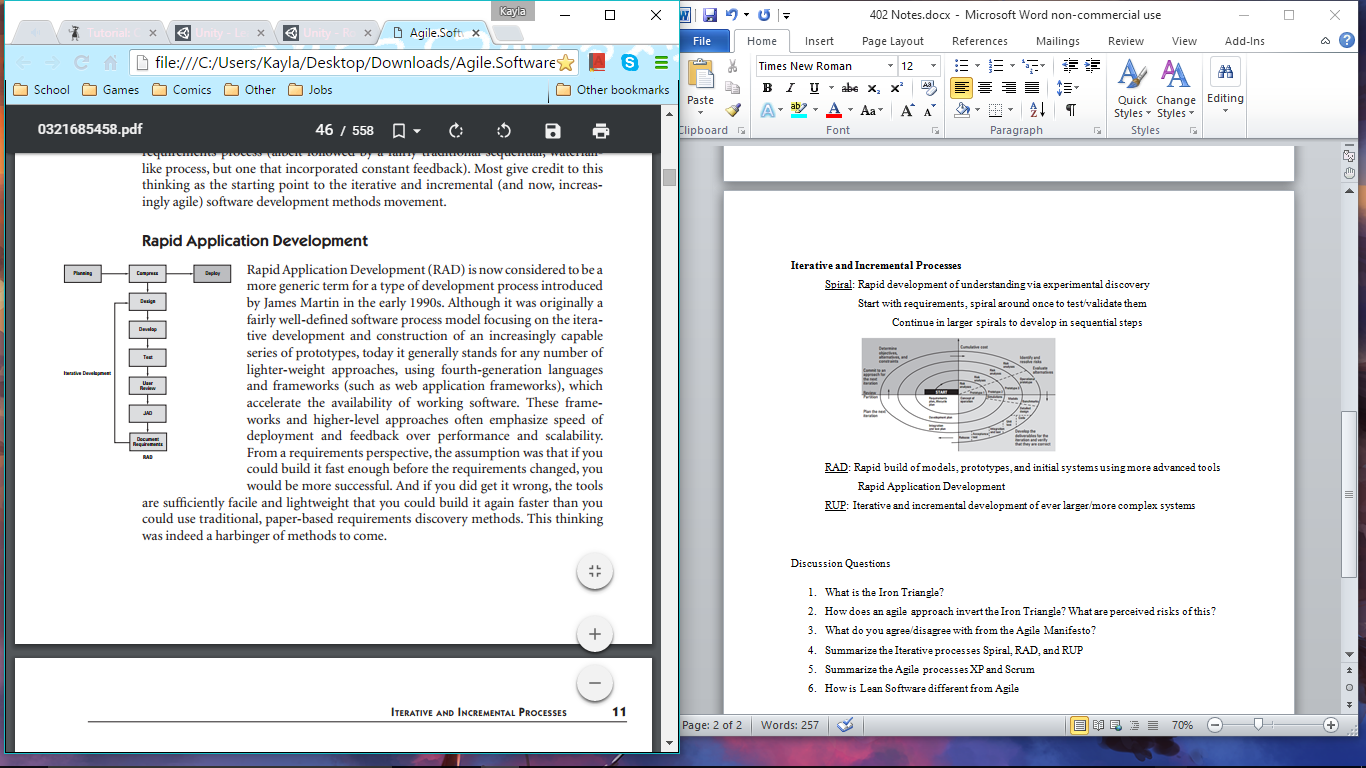
Continue in larger spirals to develop in sequential steps



RAD: Rapid build of models, prototypes, and initial systems using more advanced tools

Rapid Application Development: build fast before the requirements change

Emphasize speed of deployment/feedback over performance/stability

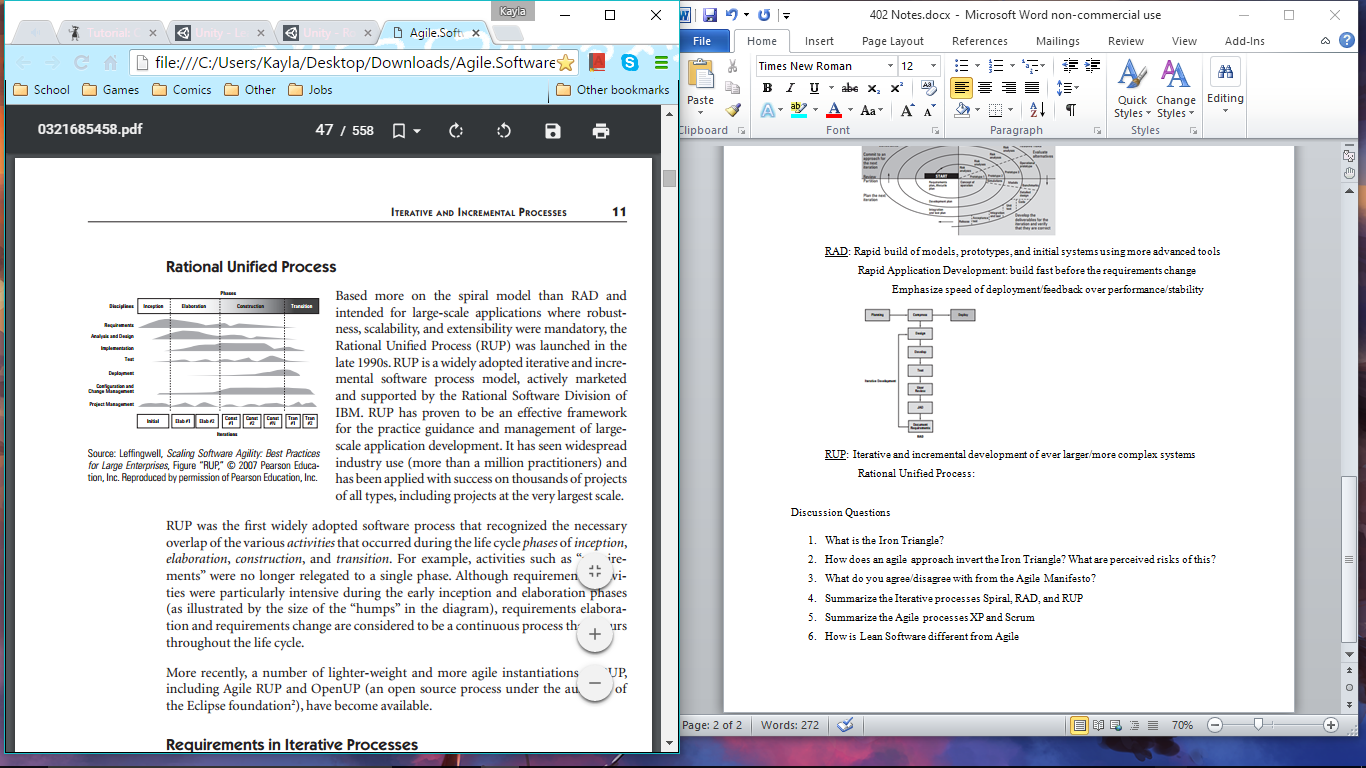


RUP: Iterative and incremental development of ever larger/more complex systems

Rational Unified Process: overlaps activities that occur during life cycle

Phases of inception, elaboration, construction, and transition

Requirements no longer in a single phase; continuous throughout life cycle



Movement away from BUFD (big, up-front design) requirements and design artifacts

Instead, discovery-based approach with lighter-weight documents/models

**Adaptive (Agile) Processes (late 90’s)**

More cost effective to write code quickly, have it evaluated by customers, then refactor

Fare more flexible approach to requirements management

More temporal, interactive, and just-in-time

The Agile Manifesto

Individuals and interactions over processes and tools

Customer collaboration over contract negotiation

Working software over comprehensive documentation

Responding to change over following a plan

Extreme Programming (XP)

Team of 5-10 programmers work at one location with customer representation

Development in frequent builds/iterations, deliver incremental functionality

Requirements specified as user stories, new chunks of user required functionality

Programmers work in pairs, follow strict standards, do their own unit testing

Requirements, architecture, and design emerge over the course of the project

Scrum

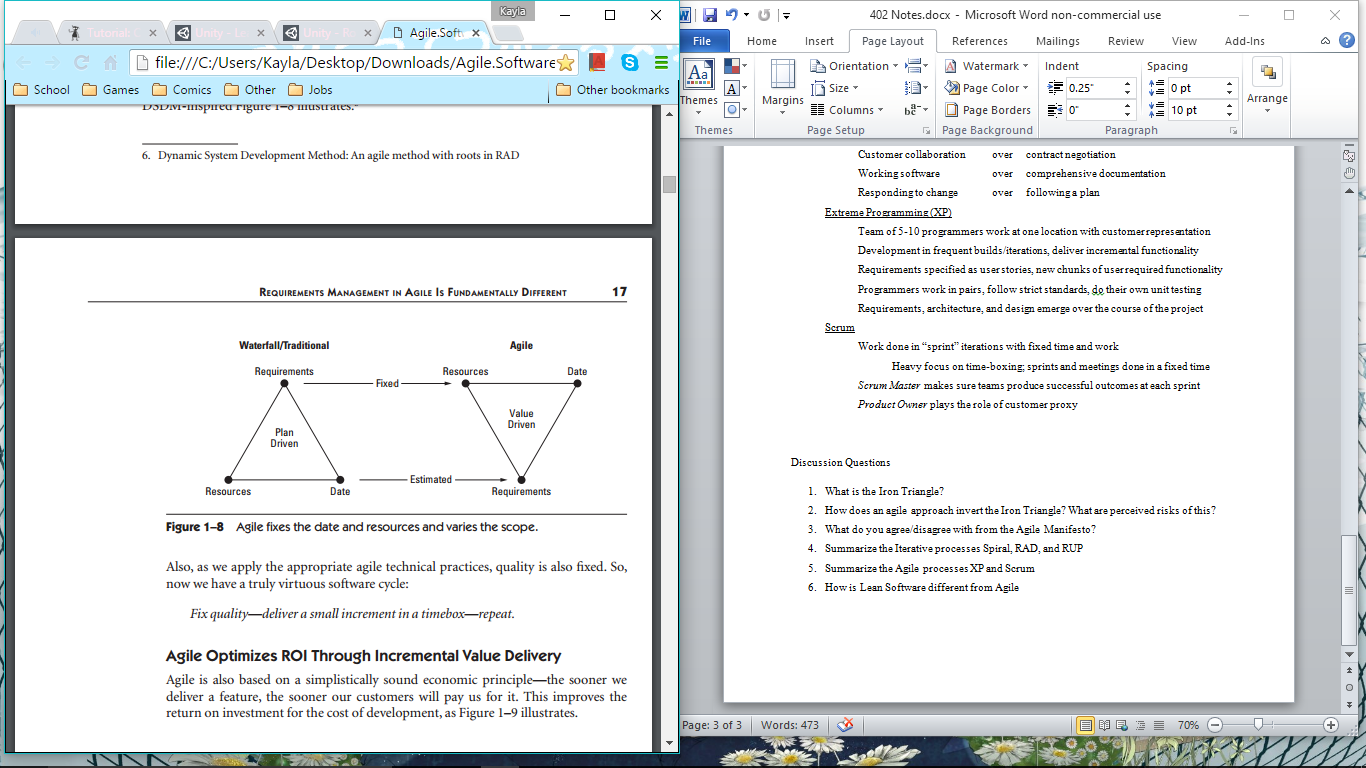
Work done in “sprint” iterations with fixed time and work

Heavy focus on time-boxing; sprints and meetings done in a fixed time

*Scrum Master* makes sure teams produce successful outcomes at each sprint

*Product Owner* plays the role of customer proxy

Inversion of Iron Triangle



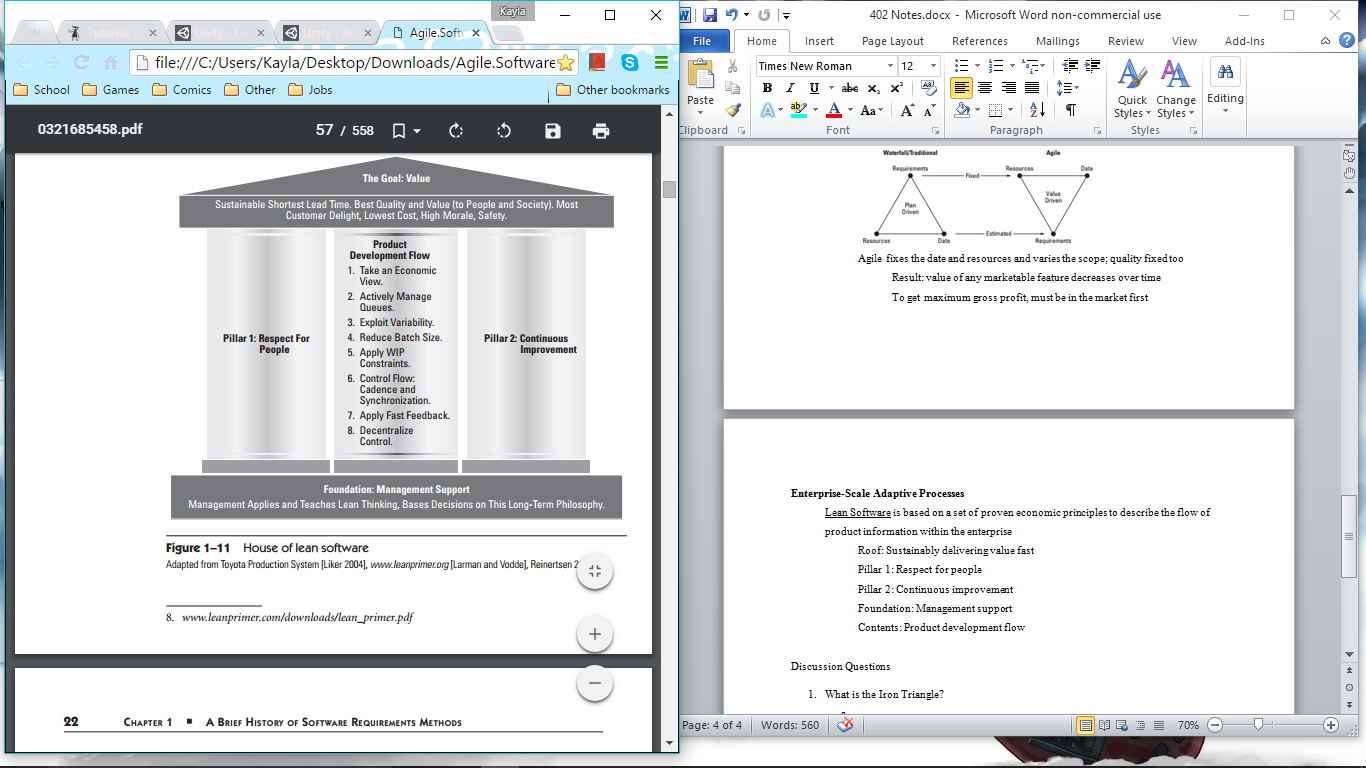
Agile fixes the date and resources and varies the scope; quality fixed too

Result: value of any marketable feature decreases over time

To get maximum gross profit, must be in the market first

**Enterprise-Scale Adaptive Processes**

Lean Software is based on a set of proven economic principles to describe the flow of product information within the enterprise



Roof: deliver maximum amount of value to customer in shortest possible time

Focus on customer requirements, rather than people who manage them

Actively minimize delays and handoffs

Pillar 1: respect for the people

People empowered to evolve/change/improve/learn by themselves

Pillar 2: continuous improvement

Solve problems/improve by going to source/personally observing

Make decisions slowly by consensus, consider options, implement rapidly

Use continuous improvement to determine root cause of inefficiencies

Protect organizational knowledge by developing stable personnel

Reflect at key milestones & identify all shortcoming of project at end

Foundation: management support

A*gile:* management supports and helps eliminate impediments

*Lean:* management leads, is competent, and takes active role in improving

Product Development Flow

Take an economic view

Establish decision framework, understand full value chain

Sequence high-risk, low-cost activities first

Actively manage queues

Manage queue lengths & improve/predict wait times

Understand and exploit variability

Instead of eliminating, design systems that expect/address variability

Reduce batch sizes

Create unnecessary variability and cause severe delays in delivery/quality

Good infrastructure, loose architectural coupling, proximity improve this

Apply WIP constraints

Apply constraints to work in progress to control queue length

Control flow under uncertainty—cadence and synchronization

*Cadence* - predictable rhythm that transforms unpredictable🡪predictable

Makes waiting times predictable, lowers transaction costs

Resynchronization limits variance

Get feedback as fast as possible

Can take fast corrective action

Truncates unsuccessful paths, reduces cost of failure in risk taking

Facilitated by small batch sizes, but more investment in environment

Decentralize control

Faster we go, less practical to have decisions move in chain of command

**Discussion Questions**

1. What is the Iron Triangle?
2. How does an agile approach invert the Iron Triangle? What are perceived risks of this?
3. What do you agree/disagree with from the Agile Manifesto?
4. Summarize the Iterative processes Spiral, RAD, and RUP
5. Summarize the Agile processes XP and Scrum
6. How is Lean Software different from Agile?

**[Intro to Software Requirements Sept. 24](#TableOfContents)**

What question do software requirements answer?

Primarily answer what – what is the system to do?

Who – who are the system user groups?

Business case tells us why

Project plan tells us when and who

Architecture tells us how

Cost of avoiding/fixing defects increases as project progresses

Reason why Waterfall Model lasted so long

Paranoid about catching defects early, so made all requirements upfront

IEEE Definition of Requirement

A condition or capability needed by a user to solve a problem or achieve an objective

A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document

A documented representation of a condition or capability as in 1 or 2

Types of Requirements

Business

High-level objectives of the organization or customer who requests system

Documented in a Vision and Scope document

User

User goals or tasks that the users must be able to perform with the product

Use-cases or user stories often used to capture these

System

High-level requirements for a product that contains multiple subsystems

Functional

Specify the software functionality that the developers must build into the product to enable users to accomplish their tasks

Ex) The system *shall* mail a confirmation to the user

Non-Functional

Quality attributes, performance goals, reliability

**[Chapter 2: The Big Picture of Agile Requirements Sept. 28](#TableOfContents)**

**The Team Level**

Agile teams of 7±2 define, build, and test *user stories* in *iterations* and *releases*

Organized to deliver software *features* or *components*

Larger enterprises have 3-10 teams (pods) cooperate to build larger features

Supported by architects, external QA resources, specialists, etc.

Product Owner determines/prioritizes user requirements and maintains product backlog

Works with stakeholders (customers/analysts) to determine requirements

Sets objectives for the iteration, maintains backlog and priorities

Elaborates stories, participates in progress reviews, accepts new stories

Scrum Master is the team-based leadership proxy who assists team in transitions to new methods and facilitates a team dynamic to maximize performance

Facilitates team’s progress towards the goal

Leads the team’s efforts in continuous improvement

Enforces the rules of the agile process

Eliminates impediments

Developers/Testers write and test the code; limited to ~3-4 developers and ~1-2 testers

Developers: collaborate with owners/testers to make sure right code is developed

Write the code, write/execute unit test, check code into shared repository

Testers: interface with developer/product owner to make story is understood

Write acceptance test case, test against acceptance, check into repository

New functionality is built in short time-boxed events called iterations

Plan iteration, build/test stories, demonstrate to stakeholders, inspect/adapt, repeat

Recommended length of 2 weeks; large enterprises adopt a standard length

User stories are brief statements of intent describing something the system needs to do for some user

“As a <user role>, I can <activity> so that <business value>”

Team Backlog consists of all the user stories the team has identified for implementation

Maintained and prioritized by the team’s product owner

Tasks must be accomplished by individual members in order to complete a story

**The Program Level**

Large-scale system development done by multiple teams in an Agile Release Train

Time-boxed iterations/milestones that are date/quality fixed, but variable scope

Produces *releases* or *potentially shippable increments* at frequent intervals

Product manager responsible for defining features of the system (vision)

Own the vision and program (release) backlog

Manage release content, maintain the product Roadmap

Build an effective product manager/product owner team

Releases and Potentially Shippable Increments (PSI)

May build up technical debt, not every increment is shipped to customer

Release planning: break features into stories, negotiate scope trade-offs

Roadmap updated by release planning and provides sense of how to deliver increasing value over time

Consists of a series of planned release dates

Committed to the enterprise and its current “plan of intent”

Vision

Primary content is a prioritized set of *features* intended to deliver *benefits* to users

Contains nonfunctional requirements (reliability/accuracy/quality/standards, etc.)

Undelivered features fill the program backlog

**The Portfolio Level**

Investment themes establish the relative investment objectives for the enterprise/business

Key-product value propositions that provide marketplace differentiation

Have longer life spans than epics; may be unchanged for up to a year

Drive the portfolio vision for all programs, and derive new epics from the themes

Epics represent highest-level expression a customer needs

Developing initiatives intended to deliver the value of an investment theme

Identified, prioritized, estimated, and maintained in the portfolio backlog

Decomposed into specific features prior to release planning

Objective is strategic intent, not specificity

Architectural Runway

The ability to implement new features without excessive refactoring

Teams who can do this will eventually emerge as the winners in the marketplace

**Discussion Questions**

1. What are the typical outcomes of the team, program, and portfolio levels?
   1. Team: user stories
   2. Program: releases and potentially shippable increments
   3. Portfolio: investment themes, epics, portfolio vision
2. What is an epic?
   1. An epic is a developing initiative that is intended to deliver the value of an investment theme, and is identified/prioritized/estimated/maintained in the portfolio backlog
3. What is the difference between the Product Owner and the Scrum Master in an Agile Team?
   1. Product owner determines/prioritizes user requirement/maintains product backlog
   2. Scrum master is the team-based leadership proxy who assists team in transitions to new methods and facilitates a team dynamic to maximize performance
4. What is the difference between a User Story and a Task?  Which goes on the Product Backlog?
   1. User stories are brief statements of intent, describing something the system needs to do for some user
   2. Tasks are things that must be accomplished by individual members in order to complete a story
   3. Product backlog consists of all the user stories the team has identified for implementation
5. What is an Architectural Runway?
   1. The ability to implement new features without excessive refactoring.
6. Where does the Vision and Scope document fit in all this?
   1. Vision and Scope are maintained by the product manager in the program level.
7. Where does the SRS fit in all this?
   1. SRS is contained in the vision and scope.

**[Chapter 3: Agile Requirements for the Team Sept. 28](#TableOfContents)**

Team Level

Teams organize around requirements to optimize efficiency of defining/building/testing

Basic unit of work is the user story

Involved in defining requirements, optimizing requirements/design trade-offs

Define/Build/Test sequence

Define: even if the story is well-elaborated, developer will likely interact with product owner to understand what is meant by story

Combination of both requirements and design; inseparable

Build: actual coding of the story

Story understanding evolves during coding process

Test: story is not completed until it has passed an acceptance test, which assures code meets the intent of the story

Built before or in parallel with code to test team’s understanding of story

Product backlog holds all work (primarily user stories) that needs to be done

Is local to the team and is managed by them

Tasks represent activities that must be performed by specific team members to accomplish the story

Have an owner (person responsible for it) and are estimated in hours (usually 4-8)

Stories are implemented by tasks

Ron Jeffries User Story

Card represents 2-3 sentences used to describe intent of the story

Conversation represents fleshing out details of the intent of the card

Done in a conversation with the customer/product owner

Confirmation represents how team understands the code meets full intent of story

Acceptance Test confirms the story has been implemented correctly

Story Acceptance Test; done when it passes

Unit Tests confirm that a function or procedure works as intended

**Discussion Questions**

1. How does an Agile Team divide the tasks of define, build, and test with a user story?
   1. Tasks are all done in parallel and organized around the requirements
   2. Define: even if the story is well-elaborated, developer will likely interact with product owner to understand what is meant by story
      1. Combination of both requirements and design; inseparable
   3. Build: actual coding of the story
      1. Story understanding evolves during coding process
   4. Test: story is not completed until it has passed an acceptance test, which assures code meets the intent of the story
      1. Built before or in parallel with code to test team’s understanding of story
2. How are Product Owners, Scrum Masters, Developers and Testers in Agile Teams different from Requirements Analysts, Project Managers, Developers and Testers in Plan-Driven Teams?
   1. In plan-driven teams product managers, business analysts, developers and program managers are compartmentalized and do not interact with each other.
   2. In agile teams are redefined and silos that separate functions are eliminated.
      1. Cooperate with other team members and check to make sure stories and requirements are understood
3. How do we keep "Other Work Items" as Stories and not Tasks?
   1. Work items are called stories because that’s what most agile teams and tools call them.
   2. Tasks are the individual assignments needed to complete a story
4. What is the difference between Acceptance Tests and Unit Tests?
   1. Acceptance tests confirms the story has been implemented correctly
   2. Unit tests confirm that a function or procedure works as intended
5. How does TDD fit into an agile process?
6. What is the standard form of a user story?
   1. “As a <user role>, I can <activity> so that <business value>”

**[Chapter 12: Requirements Discovery Toolkit Sept. 28](#TableOfContents)**

Requirements discovery represents how teams should go about understanding what problems their solution is intended to address, what markets or types of customers it is intended to serve, and what the functional and nonfunctional requirements for such a system needs to be.

Discovered through requirements workshops, brainstorming, interviews/questionnaires,

User experience mock-ups, product council, competitive analysis, use-case modeling

Requirements workshop drives consensus on requirements of system and gains rapid agreement on a course of action from key stakeholders in a very short time

Key stakeholder are gathered for a short, intensive period (1-2 days)

Forges agreement between stakeholders/p.owners/dev. team about what to do

Assists in building effective team of stakeholders, committed to a common vision

All stakeholders get their say; no one is left out

Can expose/resolve political issues that may interfere with project success

Workshop facilitated by product owner/product manager/team member/facilitator

Preparing for the workshop:

(1) Sell the concept

(2) Ensure participation of the right stakeholders

(3) Attend to logistics

(4) Provide warm-up materials

(5) Set role of the facilitator

Set the agenda

Running the workshop:

Brainstorming and idea reduction

Encourage participation, “piggyback” on other ideas, out-of-box thinking

Idea generation

Facilitator explains objective of the process, asks questions

Ideas are written down by person who came up with them Idea reduction

Pruning ideas, grouping ideas, defining features

Idea prioritization

Cumulative voting ($100 test)

“Critical/Important/Useful” categorization

Critical = mandatory

Important = loss of customer utility/market/revenue

Useful = nice to have; more appealing

Online brainstorming

Summary of requirements workshop and brainstorming

Production and follow-up

Interviews and Questionnaires

User/Stakeholder interview used to gather requirements

Makes sure biases/predispositions don’t interfere with free exchange of info

Context-Free Questions

Ask ?s about the nature of user’s problem without context for potential solutions

Ex: Who is the user? Who is the customer? Are their needs different?

Solutions-Context Questions

Begin exploring potential solutions

Interview:

Prepare an appropriate context-free interview

Understand background of the stakeholder to be interviewed

Jot down answers in notebook

Compile the needs data

Analyst’s summary: record 3 most important needs/problems in interview

Questionnaires: do **not** substitute interviews with these

User Experience Mock-Ups

Form a Product Council

Empowered to act as the approval body on key decisions

Should consist of key stakeholders in the enterprise/business unit

Competitive Analysis

Refine the domain of interest/product category

Identify competitors

Study competitive offerings

Prepare an analysis of the finding for evaluation

Customer Change Request Systems

Defect logs

Use-Case Modeling

**[Chapter 13: Vision, Features, and Roadmap Sept. 28](#TableOfContents)**

Vision is used to continuously communicate the strategic intent of the program

Vision document: prompt of the things that need to be described to teams for them to understand what they are about to build, who uses it, and how they use it to do their jobs

Why are we building this product, system, or application?

What problem will it solve?

What features and benefits will it provide?

For whom does it provide these features and benefits?

What performance, reliability, and scalability must it deliver?

What platforms, standards, applications, and so on, will it support?

Advanced Data Sheet: extremely concise document (2 pages), focuses on what is critical to communicate

Features are the primary content of the vision

Product roadmap is a tool used to communicate how the future of the program will unfold

**[Requirements Elicitation Sept. 29](#TableOfContents)**

Break features into use cases and user stories

Ex) Ride sharing is a feature

User stories: “As a <user role>, I can <activity>, so that <business value>”

Use cases: Login

Exceptions and alternative courses

Functional and Nonfunctional requirements

Functional requirements: “The system shall \_\_\_”

Discovering user requirements

Help users figure out what they want

Finding requirement process:

Do outside research in domains of concern

Only speak user’s terms and definitions

Ask questions (first few rounds)

Context-free questions (no bias of a solution)

After initial rounds, ask questions with binary answers (yes/no)

Analyze, follow up, repeat

Elicitation Interviews

Do basic research first

Do **not** ask questions that have been answered

Follow-up on previous sessions

Focus your questions

Beware of broad questions

Break complex questions into multi-part questions

Use models/documents as points of reference

Give feedback on the answers

Offer examples, narrow the question

Do not move on until understand/agree to look further

Think like a customer and like a coder!

Artifacts

User stories, use cases, functional/nonfunctional requirements

Analyze customer’s answers

Give them an identification number, file with all relevant information

Carefully parse answers for critical info

Priorities, definitions of new domain terms

Generate new questions if necessary

Passive or Active Elicitation

**[Chapter 6: User Stories Oct 3](#TableOfContents)**

A user story is a brief statement of intent that describes something the system needs to do for the

user

Written by *customer* in XP

Written by *product owner* in Scrum, w/ input from customers/stakeholders/team

Any team member w/ sufficient domain knowledge can write user stories

Up to product owner to accept/prioritize potential stories into product backlog

Tool for defining system’s behavior in a way understandable to developers and users

Focus on the value defined by user rather than functional breakdown structure

Short statement of function or list of things the system needs to do for the user

User Story Form

Card, Conversation, and Confirmation

Card = 2-3 sentences describing intent of the story

Conversation = discussion between team/customer/product owner/stakeholders

Confirmation = conditions of satisfaction applied to determine whether story

fulfills the intent as well as the more detailed requirements

Acceptance tests

User Story Voice

As a <role>, I can <activity>, so that <business value>

User-first perspective, enhances the “why” and “how” understanding

User Story Detail

Conveyed through conversations between product owner and team

User Story Acceptance Criteria

Additional notes and assumptions that are kept with the story

Conditions of satisfaction

Many of these can be turned into acceptance test cases for the story

INVEST in Good User Stories

**I**ndependent **N**egotiable **V**aluable **E**stimable **S**mall **T**estable

Describe the attributes of a good user story

Independent: can be developed/tested/delivered on its own. Deliver value independently

Negotiable: placeholder for requirements to be discussed/tested; helps predictability

Valuable: deliver the most value given existing time/resource constraints; prioritized

Estimable: able to provide approximate estimation of complexity and required work

If not, too large (split into smaller stories) or uncertain (technical/functional spike)

Small: smaller stories increase *throughput* and *decrease complexity*

Increased throughput

Cycle time = Work in process / Throughput

Fewer, smaller stories in process will come out faster

Variation in cycle time decreases, meaning more predictable

Decreased complexity

Do one thing, keep them small, make them smaller than that

Testable: all stories must be testable

Splitting User Stories

Compound stories are usually too big to be implemented within an iteration

Break them down into smaller stories using the ten common patterns

Workflow Steps

Include specific steps a user takes to accomplish a specific workflow, then

implement the workflow in incremental stages

Business Rule Variations

Break the story into several stories to handle the business rule complexity

Major Effort

Story can be split into several parts where most of the effort will go toward

implementing the first one

Simple/Complex

Capture the simplest version that can possible work, then break out all variations

and complexities into their own stories

Data Entry Methods

Split the story to build it with the simplest possible UI, then build richer UI later

Defer System Qualities

Break the story into successive “ilities”

Operations

Words like *manage* or *control* giveaway that the story covers multiple operations, which can offer a natural way to split the story

Use-Case Scenarios

If use cases have been developed to represent complex interactions, then story can

be split according to the individual scenarios of the use case

Break Out a Spike

Build a technical/functional spike to figure it out, then split stories on that result

Spikes

Spikes are a special type of story used to drive out risk and uncertainty in a user story

Spikes are used for a number of reasons:

May be used for basic research to familiarize team with new technology/domain

Story may be too big to be estimated appropriately; analyze implied behavior to

split the story into estimable pieces

Story may contain significant technical risk; team has to do research/prototyping

to gain confidence in a technological approach that will allow them to commit

Story may contain significant functional risk

Technical spikes are used to research various technical approaches in the solution domain

Functional spikes are used whenever there is significant uncertainty as to how a user might interact with the system

Best evaluated through some level of prototyping

Ex) As a consumer, I want to see my energy use in a histogram so I can …

Tech: research how long it takes to update a customer display to current usage, determining communication reqs., bandwidth, and whether to push/pull data

Func: Prototype a histogram in the web portal and get some user feedback on

presentation size, style, and charting attributes

Guidelines for Spikes

Estimable, demonstrable, and acceptable

The exception, not the rule

Implement the spike in a separate iteration from the resulting stories

**Discussion Questions**

1. What is a user story?
2. Who writes user stories?
3. What is the recommended form of a user story?  Why is this recommended?
4. Give an example user story and corresponding acceptance criteria.
5. Expand and describe the INVEST acronym in regards to good user stories.
6. Briefly summarize each of the ten patterns for splitting a user story.
7. Is a vertical prototype more like a technical spike or a functional spike?

**[Chapter 7: Stakeholders, User Personas, and User Experiences Oct. 3](#TableOfContents)**

Users are key project stakeholders

**System Stakeholders**

A system stakeholder is anyone who directly uses the system, works w/ results of those who use the system, or will be impacted by the deployment and operation of a system

Users, operators, data users, managers, purchasers, admins, maintenance, help…

Primary drivers for the requirements of the system

**Project Stakeholders**

A project stakeholder is anyone who has a vested interest in the budget/schedule, has a vested interest in understanding how the product is developed, or is involved in marketing/selling/maintaining the system

People who have a substantial interest in the project that is developing the system

**Product Owner**

Serves as the primary representative to all the stakeholders

Diverse requirements must *aligned* and expectations *managed* toward a solution

Merge stakeholders into a single prioritized backlog for the team

Facilitating: facilitator of a process to converge opinions into a single product vision

Leading: makes decisions for stakeholders based on expert knowledge/experience

Stakeholder Involvement:

They should be kept informed

They should be consulted

They are partners in development

They are in control of outcomes

Stakeholder Interactions:

Timely decisions are needed

Active participation is needed

Teams must take an enterprise view

Production and support staff should be involved from the start

Plan for system maintenance

Identifying Stakeholders

Project Stakeholders

Who needs to be consulted on scope? Who has input to budget/schedule?

Who manages business relationships? Determines when/how system released?

Who can support/harm project? Who is dependent on system?

Understanding project stakeholder needs

Product characteristics

Project characteristics

Partner

Expects interfaces to remain the same, stability, backwards compatibility

Wants input in features, notification of changes in schedule/priority

Sales/marketing

Expects features available, roadmap, release when/how, product benefits

Wants input in features, notifications of priority change/delays

Operations

Expects installation/dependency details, reliability and performance

Wants project statuses, involvement in installation/dependencies

Support

Expects quality/support, error management system, quick issue resolution

Wants project status updates, input in prioritization of support issues

Sponsor

Wants team to have effective processes for understanding requirements

Wants to be continually apprised of schedule/budget/status

Development management

Wants to know system will be fit for intended purpose

Expects accordance with budget/schedule guide, strategic investments

Security

Expects relevant security standards, security coding/testing practices

Consult about security issues, review designs for security flaws, input

System Stakeholders

Who will be using system/results of system? Who will support the system?

What interfaces must be provided? What other systems will it interact with?

Who can provide guidance on functionality and system qualities?

Classifying system stakeholders

First degree: direct users of the system

Second degree: people who work w/ results of those who work w/ system

Third degree: people who install/deploy/support the system

Understanding system stakeholders needs

System characteristics: what do they expect from the system?

System activities: what do they need to do with the system?

User Personas

User personas provide a means of further refining the approach to the user to make sure that the needs of different types of users are met

Primary personas represent users with specific needs that can be satisfied only with a

user interface designed specifically for them

Secondary personas also use the system but can use an interface for a primary persona

Agile and User Experience Deployment

User Experience Problem

UX design is complicated when user experience testing is required

Scheduling/running of multiple UX tests cannot occur within an iteration that is also attempting to complete the story

Results in delayed feedback loops that complicate productivity and add delays

Low-Fidelity Options for User Interface Deployment

Simple HTML/PowerPoint prototypes, mock objects, coded skeletons, sketches

Leverage iterations to drive out uncertainty and risk through fast feedback

Centralized User Experience Development

Central user interface design team that iterates independently from dev. teams

Backlog contains UX story spikes, UX testing, prototyping, etc.

Work 1 or 2 iterations ahead to discover upcoming functionality

Increases core competence in UX domain and prevents “wagging”

Distributed, Governed User Experience Development Model

Small, centralized UX design authority

Provides basic design standards + preliminary mock-ups for each UI

Provides HTML designs, CSS style sheets, brand control, mock-ups, etc.

Attends iteration+PSI/release demos to see overall system design progress

Teams have team-based UX implementation experts for the implementation

UX experts are distributed among the teams

Doesn’t prevent refactoring, but fairly efficient process

**Discussion Questions**

1. Who are the stakeholders on our project?  Which ones are system stakeholders?  Which ones are project stakeholders?
2. What are advantages for creating low-fidelity user interface mockups?
3. Compare the centralized versus distributed user experience (UX) development models.

**[Chapter 19: Use Cases Oct. 3](#TableOfContents)**

The problems with User Stories and Backlog Items

Don’t give the designers a context to work with

Don’t give the project team any sense of scope or potential completeness

Don’t provide a mechanism for looking ahead at upcoming work

Five reasons to still use use-cases

(1) The list of goal names provides a short summary of what the system will contribute to the business and the users. It also provides a project planning skeleton.

(2) The main success scenario of the use case provides everyone with an agreement as to what the system will and will not do. Provides a context for each requirement.

(3) The extension conditions provide a framework for investigating all the little things that take up development time and budget. Provides a look-ahead mechanism.

(4) The extension scenarios provide answers to many detailed business questions.

(5) The full use case model shows that the developers have thought through every user’s needs, every goal with respect to the system, and every business variant involved.

A use case describes a sequence of actions between an actor and a system that produces a result of value for that actor

The sequence of actions describes a set of interactions between the actor and the system

Invoked when the actor provides some input to the system

The system works for the actor and executes some function, procedure, or activity

Takes its orders from the actor as to when to do what

A use case must deliver a result of value to the user

The actor is the individual or device that initiates the action

An actor is someone or something that interacts with the system

Users

Other systems or applications

A device: interface to a variety of input and output devices

**Use Case Structure**

Mandatory elements:

*Name*: describes the goal that is achieved by interaction with the actor (ex: login, load)

*Brief description*: purpose of use case described in 1-2 sentences

*Actor(s)*: lists each actor who participates with the use case

*Flow of events*: textual description of the interactions between actor and system

*Basic flow*: the main path through the use case, and

*Alternate flow*: executed only under certain circumstances

Optional elements:

*Preconditions*: conditions that must be present in order for a use case to start

*Exit conditions*: describe the state of the system after a use case has run its course

*System or subsystem*: identify whether a use case is system- or subsystem- level

*Other stakeholders*: identify key stakeholders who may be affected by the use case

*Special requirements*: special requirements that apply to the use case (ex: nonfunctional)

Building the Use Case Model

(1) Identify and Describe the Actors

(2) Identify the Use Cases

(3) Identify the Actor and Use Case Relationships

(4) Outline the Flow of the Use Cases

(5) Refine the Use Cases

Applying Use Cases

Can be used to drive incremental development, one story at a time

Keep them lightweight (no design details/GUI specs)

Don’t treat them like fixed requirements

Model them informally (use whiteboards, lightweight tools, etc.)

**[Chapter 9: Activity Diagrams Oct. 5](#TableOfContents)**

The activity state is the core symbol of the activity diagram

An activity is a state of doing something: a real-world process or the execution of a software routine

**[Use Cases and Scenarios Oct. 6](#TableOfContents)**

Use case: a set of scenarios tied together by a common user goal

Scenario: a sequence of steps describing an interaction between a user and a system

Ex) A game player places a symbol on an open square on the game board

Ex) A game player places a symbol on an occupied square on the game board

Should not be allowed; handle this situation with an alternate scenario

Actors: roles that users or systems play

Actors carry out use cases

A single user could play several roles

Multiple users could play the same role

Use case components

Name, description, steps/flow of events, pre-conditions, post-conditions

Alternate flows, exceptional scenarios

Persona

Take a person, representative of an actor who is using the system

Give them a personality and history

How does this change their perspective of the app?

Use cases are **not** the requirements!

Use Cases vs. User Stories

User stories are the different scenarios in a use case

User stories emphasize the value

Use cases emphasize the content

Use cases might be used to elaborate the details of user stories

Sometimes you need one or the other or both

**[Chapter 4: Agile Requirements for the Program](#TableOfContents)** Oct. 12

At the program level the objectives are:

*Maintaining vision and roadmap*:

*Release management*: coordinating activities to build release increments

*Quality management*: assuring results are integrated and requirements/standards are met

*Deployment*: must be managed at the program level

*Resource management*: adjust resources as necessary to address constraints/bottlenecks

*Eliminating impediments*: program leaders and managers eliminate impediments

Organizing Agile Teams

Can be categorized as feature or component approaches to organizing teams

Best answer is a mix of the two

Consider the factors and select the best strategy for your specific context

Two main factors drive the mix:

The practical limitation of the degree of specialization required

The economics of potential reuse

If not choice not obvious, optimize around co-location

Communication/team dynamics/velocity may exceed benefits of organization

If are or can be co-located, choose feature team; if not, choose component team

Component Teams

A new feature requires creation of a new backlog item for each team that contributes to it

Minimize multiplexing across features by implementing them in series, not parallel

Based on past successes, enterprise might already be organized this way

Individuals and their skills/interests/cultures/lifestyles are not interchangeable

Teams may already be co-located, simplifying communication and batch handoff

Technologies/languages may differ across components, making it difficult to do pairing

A single user feature might be a huge thing that affects hundreds of practitioners

Feature Teams

Long-lived, cross-functional team that completes many customer features 1-by-1

Teams core competence becomes the feature, rather than 1 element of the stack

Almost universally accepted way of organizing agile teams: organize around features

With agile’s focus on immediate value delivery, leans towards feature teams

Advantages:

Teams build expertise in the actual domain/usage mode of the system

Less overhead, b/c teams don’t have to pass backlog items back and forth

Team’s backlog is simplified, just one or two features at a time

Promotes the fast delivery of high value-added features

System Team

At Program lv, teams may not have all capabilities to integrate/test/deploy full solutions

An additional team complements the feature/component teams

Called a system integration, QA and deployment, release or system team

Works on the same release cadence & has a set of specific, system-level responsibilities

System-level testing

System quality assurance

System-level continuous integration

Building development infrastructure

Release Management Team

Agile teams may not have the visibility/quality assurance/release governance authority to

decide when and how the solution should be delivered to the end users

Members include key stakeholders of the Program level of the enterprise

Team meets weekly to address the following questions:

Do all teams still understand their mission?

Do they understand what they are building?

What is the status of the current release?

What impediments must we address to facilitate progress?

Are we likely to meet the release schedule, and if not, how do we adjust scope?

Product Management

Set of stakeholders who have same role as product owner, but for solution as a whole

Responsible for end-to-end solution: content of release, distribution, documentation, etc.

Vision

Addresses larger ?s: What is the intent? What problem will it solve? Features/benefits?

Vision document, draft press release, preliminary data sheet, backlog and vision briefing

Features: services provided by the system that fulfill stakeholder needs

Live at a level above software requirements; bridge gap from problem to solution domain

Realized by stories; at release-planning, features are decomposed into stories

Typically expressed in bullet form or in a sentence or two

New features build the program backlog

Typically features require acceptance tests

Nonfunctional Requirements

Described as the system qualities: quality, reliability, scalability, etc.

Critical elements of system behavior

Must also be testable and tested; require 1 or more system qualities tests

Agile Release Train

Releases and Potentially Shippable Increments (PSIs)

Agile Release Train (ART): standard cadence of timeboxed iterations and

milestones that are date- and quality-fixed but scope variable

Produces PSIs (iterations) at frequent 60- to 120-day time boundaries

Release Planning

Periodic program activity that aligns the teams to a common mission

Translate the Vision into the features and stories needed to complete objectives

Roadmap

Consists of a series of planned release dates

Each has a theme and a prioritized feature set

**Discussion Questions**

1. Compare Component Teams versus Feature Teams as organizational structures. When is each more appropriate?  Which is most commonly used in agile teams?
2. Why would an agile organization still need a System Team?
3. What are the responsibilities of a Release Management Team?
4. What does it mean that "Features are realized by stories"?
5. Identify and describe some non-functional requirements.
6. How do we include non-functional requirements in features or user stories?

[**Chapter 5: Agile Requirements for the Portfolio**](#TableOfContents) Oct. 12

Portfolio level: governance and management model for new software asset development needs additional artifacts and higher levels of abstraction due to sheer size of the enterprise

Introduces two new artifact types: investment themes and epics

New backlog (portfolio backlog), new team (portfolio management team)

Container concepts of the portfolio vision and architectural runway

Investment Themes

Investment themes represent key product or service value propositions that provide

marketplace differentiation and competitive advantage

Portfolio Management Team

Responsible for the individual lines of business and derivation of decisions

In larger enterprises, happens on an annual/twice-annual budgeting process

Makes decisions based on some combination of:

Investment in *existing* product offerings—enhancements, support, maintenance

Investment in *new* products and services that will enhance revenue/gain market

Investment in *futures*—advanced product/service offerings that require

investment today but will not contribute to revenue until outlying years

Reducing investment for existing offers that are nearing end of their useful life

Themes have a longer lifespan than epics

Investment themes may be unchanged for up to a year or more

May be assisted by a project management office (PMO)

Epics and the Portfolio Backlog

Epics are large-scale development initiatives that realize the value of investment themes

Highest-level requirements artifacts that are used to coordinate development

Sit between investment themes and features

Typically driven by investment themes, but some can be independent

Not implemented directly; instead, broken into features, then into user stories

Epics are not directly testable; instead, tested by acceptance tests associated with

the features and stories that implement them

Epics deliver the value implied by the theme and are identified in the portfolio backlog

Portfolio Backlog

Identifies, prioritizes, estimates, and maintains epics

Expresses epics in bullet form, a sentence or two, a video, a prototype, a user interface

Mock-up, or any other form.

Architectural Runway and Architectural Epics

A system that has architectural runway has existing/planned infrastructure sufficient to allow incorporation of current/anticipated requirements without excessive refactoring

Answer to what technology initiatives need to be underway so we can reliably deliver

a new class of features

Involve significant, structural changes that could affect millions of LoC

Must be defined and communicated to the team

Business epics: functional or user-experience epics

Architectural epics: used to implement technological changes that must be made to

significant elements of the portfolio

Will be implemented in the main code line, incrementally

Development teams commit a “do no harm” refactoring approach

Implement large-scale refactors in small increments

Architectural Runway

Continuous build out/maintenance of new arch. Runway is responsibility of teams

Otherwise, miss release dates or run out of runway

Work that must happen continuously at the Portfolio, Program, and Team levels:

*Portfolio*: define/communicate/implement architecture epics

Failure will compromise company’s position in the market

Must be visible, estimated, and planned

*Program*: translate the architectural epics into architectural features

Prioritized, estimated, and resourced

Must be conceptually complete at each release boundary

*Team*: refactors/design spikes extend the runway

Prioritized along with user stories

Visible, accountable, and demonstrable at every iteration boundary

**Discussion Questions**

1. What is the difference between Investment Theme, Epic, Feature, and Story?
2. How does Architectural Runway enable new features to come out faster?
3. Can you think of an example of how you can include architectural epics in feature development?

**[Chapter 17: Nonfunctional Requirements](#TableOfContents)** Oct. 12

Nonfunctional requirements describe the system qualities

Constraints on a new development (backlog constraints)

Ex) Security, reliability, scalability, etc.

A backlog item may be constrained by 0, 1, or more nonfunctional requirements

A nonfunctional requirement may apply to 0 or more backlog items

Affect the overall usefulness and viability of the solution

(F)**URPS**

Functionality: what the system does for the user

Usability: how easy it is for a user to get the system to do it

Specify the training time objective for a user to become minimally productive

Specify measurable task times for typical tasks or transactions

Compare the user’s experiences with other comparable systems

Specify any required user assistance features (online help, manuals, etc.)

Reliability: how reliably the system does it

*Availability*: available for operational use during a specific % of time

*Mean time between failures (MTBF)*: specified in units of time (hours/years/etc.)

*Mean time to repair (MTTR)*: how long the system out of operation after failure

*Accuracy*: what accuracy is required in systems (to the penny? Dollar?)

*Defects*: categorized in terms of minor, significant, and critical

*Security*: detect denial-of-service attacks, etc.

Performance: how often, or how many of it, it can do

*Response time*: specify for transactions of a given type, average vs. worst case

*Throughput*: specify in transactions per second, latency, overhead, data rates, etc.

*Capacity*: specify # of customers/transactions/data system can accommodate

*Scalability*: specify ability of the system to be extended to accommodate more

*Degradation modes*: behavior for when the system has been degraded

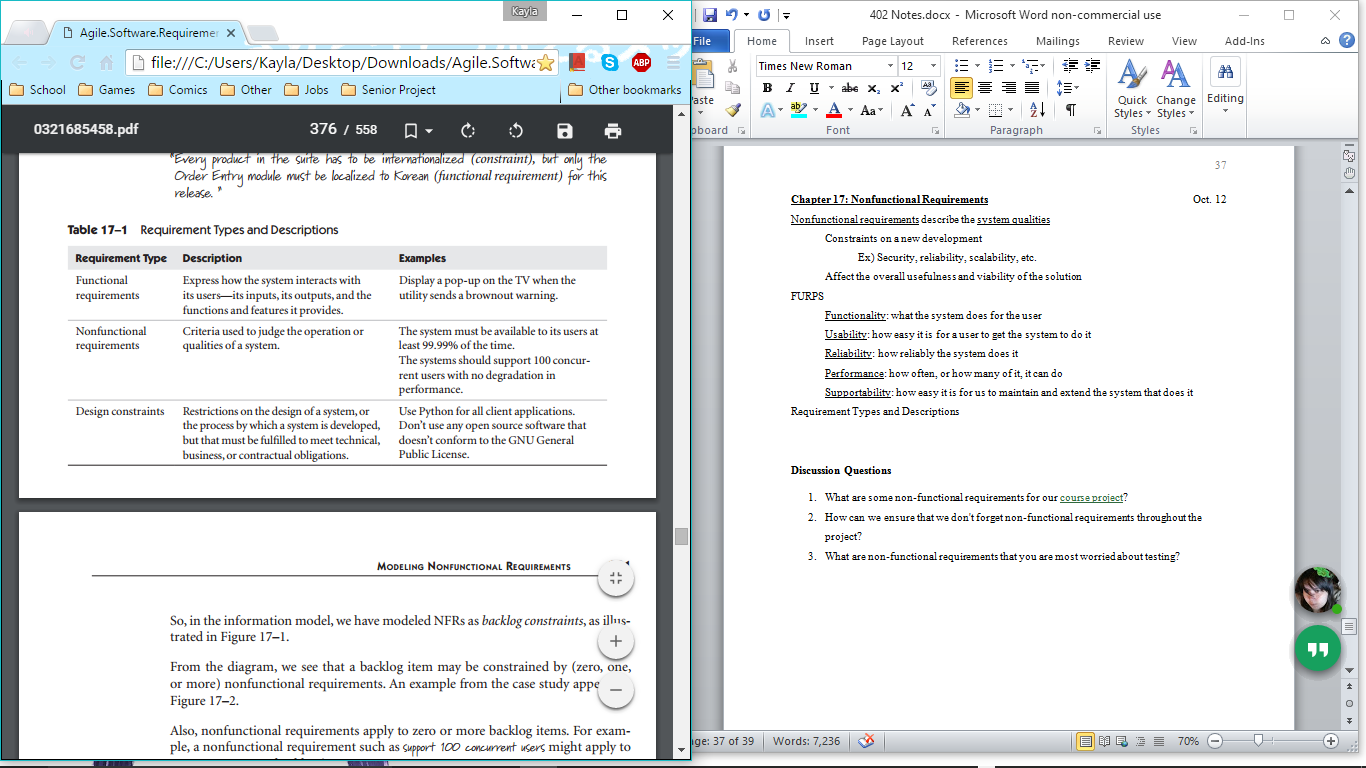
*Resource utilization*: degree to which system will share hardware resources

CPU, memory, channels, disk storage, bandwidth, etc.

Supportability: how easy it is for us to maintain and extend the system that does it

Ability to be easily modified to accommodate enhancements and repairs

Requirement Types and Descriptions



Design Constraints

Restriction of design options

Conditions imposed on the development process

Application standards

Corporate best practices and standards

Development standards

Regulations and imposed standards

Managing design constraints

Keep in a special section of backlog, identify source and rationale

Nonfunctional requirements may persist throughout the development life cycle

Create a separate backlog in the agile project management tool

Store and manage them in a wiki

Maintain a supplementary specification

Builds the NFRs into the definition of done, and point to the backlog/wiki/specification

that contains the details

Testing Nonfunctional Requirements

Not all have quality tests, but most should have at least one objective test associated

Every system quality test should be associated with some NFR

Usability: productivity, accuracy, recall, emotional response

Reliability: easy because objectives are generally clearer

Might not be possible to test product for the amount of time necessary to get data

Could be difficult to simulate user’s real operating environment (other factors)

Security:

White-box testing: examines code to look for potential coding practices/paths that

allow security breaches

# of potential combinations must be pruned to manageable set of tests

Black-box testing: mimics the way real-life hackers try to defeat a system

Use scripts and tools to inject various faulty inputs into the system

Try to “break” the system like a hacker would

Performance: done with specialized tools (simulators, measuring/monitoring tools)

Supportability and Design Constraints:

Tests adherence to any supportability requirements that may be imposed on team

**Discussion Questions**

1. What are some non-functional requirements for our [course project](https://polylearn.calpoly.edu/AY_2015-2016/mod/assign/view.php?id=73834)?
2. How can we ensure that we don't forget non-functional requirements throughout the project?
3. What are non-functional requirements that you are most worried about testing?

**[In-Class Presentations](#TableOfContents) Oct. 6**

Balsamiq (Mallika)

Rapid iteration wire-framing tool

Features:

Add elements quickly

Drag and drop UI library

Autocomplete sample text

Easy helper cues

Low fidelity wireframes

Lightweight in looks and effort

Really low-fidelity; keep it in a flexible, creative zone

In-between tool

Zen area, rough approach (not final)

Large prototypes are bad, simplistic on purpose

$429 for 5-person license, with a freemium option for education

Very small company (less than 10 employees)

Available for both web and desktop

Invision (Erica)

Prototyping, collaboration and workflow platform for designers

Basic screen designs + gestures/transitions/hotspots = interactive

Share project link with clients/other team members

Can be opened on desktop or mobile device

Clients and stakeholders can mark directly on the designs

Other team members can collaborate at the same time

Looks and works just like a real app would (mimics the app)

Requires external creation of mock-ups (no option in Invision to make prototypes)

Free to use

Violet (Connor)

UML diagramming

Class, sequence, state, object, use case diagrams

Advantages: easy to learn and use, free and open source, cross platform

Drawbacks: does not generate code, no reverse engineering, no import/export to other UML diagramming tools

Can export as images or as HTML

CMMI (Cody)

Capability Maturity Model Integration

Framework for performance improvement to achieve high-performance operations

5 levels of maturity

Started in 1986 because the military was over budget and slow

Wanted to make their software system better and more efficient

Hired Carnegie Melon to improve their performance

3 different types of CMMI models

Acquisition: assemble a product/deliver service

Development:

Services:

Take in a lot of information and decide which level of maturity a company is

1 – initial, 2 – managed, 3 – defined, 4 – quantitatively managed, 5 – optimized

Generally only used by military or older companies

Need to prove that they are safe and can get out reliable products