An Introduction to Object-Oriented Analysis and Design and the Unified Process

"Applying UML and Patterns, 3rd ed." – Craig Larman, pp. 1 – 100

Kakarontzas George
gkakaron@teilar.gr

Important skills for an object-oriented developer

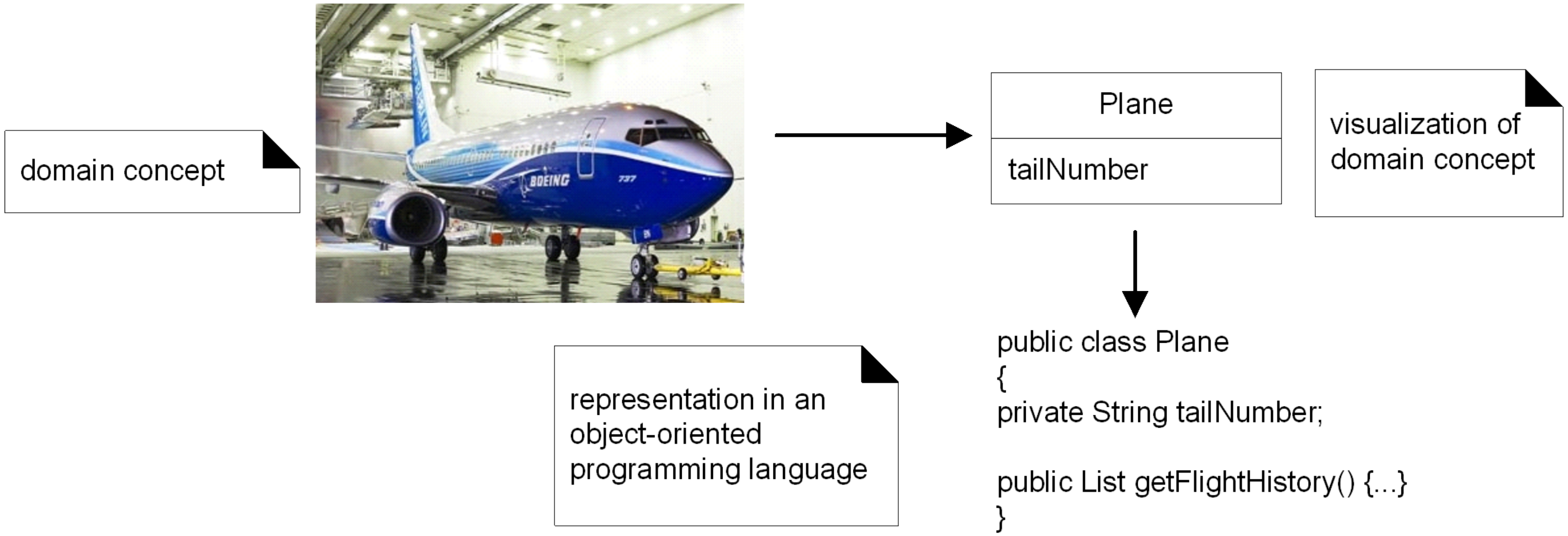
- Object-Oriented Analysis and Design (OOA/D)
- UML
- Requirement Analysis
- Design Patterns
- Principles and Guidelines
- Iterative Development
- The most basic skill: “Know how to assign responsibilities to objects”

Analysis and Design

- Analysis: *Do the right thing*
  - Investigate what is the problem and what are the requirements, rather than a solution.
  - A broad term, best qualified, as in *requirements analysis* or *object-oriented analysis (what are the domain objects?)*
- Design: *Do the thing right*
  - Design a conceptual solution to the problem, omitting obvious implementation details
  - Again a broad term, best qualified, as in *object-oriented design, database design etc.*

Object-Oriented analysis and design

- Object-Oriented Analysis: Discover the domain concepts (the objects of the problem domain)
- Object-Oriented Design: Define software objects and how they collaborate to fulfill the requirements


domain concept
Plane
tailNumber
visualization of domain concept
representation in an object-oriented programming language

```
public class Plane
{
private String tailNumber;

public List getFlightHistory() {...}
}
```

What is UML

- “The Unified Modeling Language (UML) is a visual language for specifying, constructing and documenting the artifacts of systems. It is a general-purpose modeling language that can be used with all major object and component methods, and that can be applied to all application domains (e.g., health, finance, telecom, aerospace) and implementation platforms (e.g., J2EE, .NET) [Object Management Group: “UML 2.0 Infrastructure Specification”]

What UML isn’t

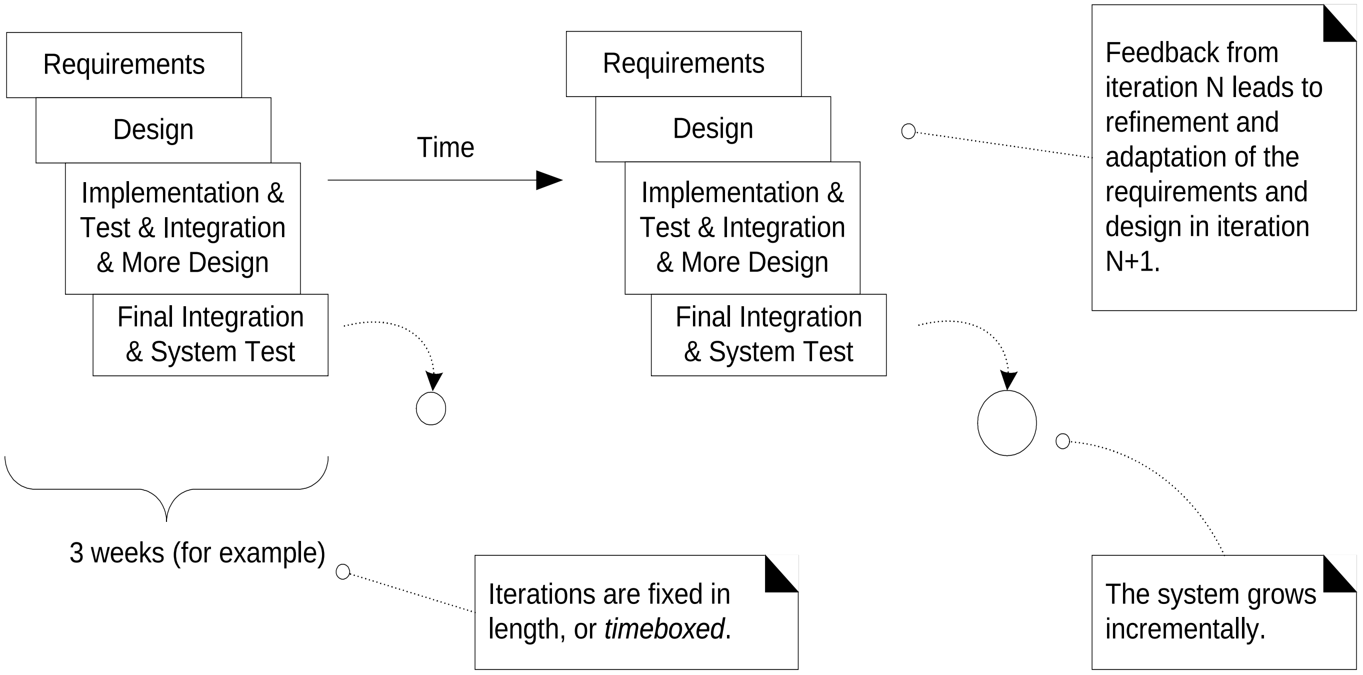
- UML is not an Object-Oriented Analysis and Design process (i.e. a systematic way to develop software systems).
- UML will not teach you an Object-Oriented way of thinking:
  - It will not tell you how to assign responsibilities to objects or whether your design is good or bad.
- UML is not meant to be the complete solution for your software development (unless you use it as in Model-Driven Architecture – MDA)

Software Development Processes and UP

- A software development process describes an approach to building, deploying and possibly maintaining software.
- The Unified Process (UP) has emerged as a popular *iterative* software development process for building object-oriented systems.
- UP:
  - Is iterative
  - Provides an example structure for OOA/D
  - Is flexible (can be combined with practices from other OO processes)

Iterative Development

- Development is organized in a series of fixed-length iterations (mini-projects).
- Each iteration includes it's own requirement analysis, design, implementation, integration and testing.
- This method is also:
  - Incremental: system grows with each iteration.
  - Evolutionary: feedback from each iteration evolves specification and design.

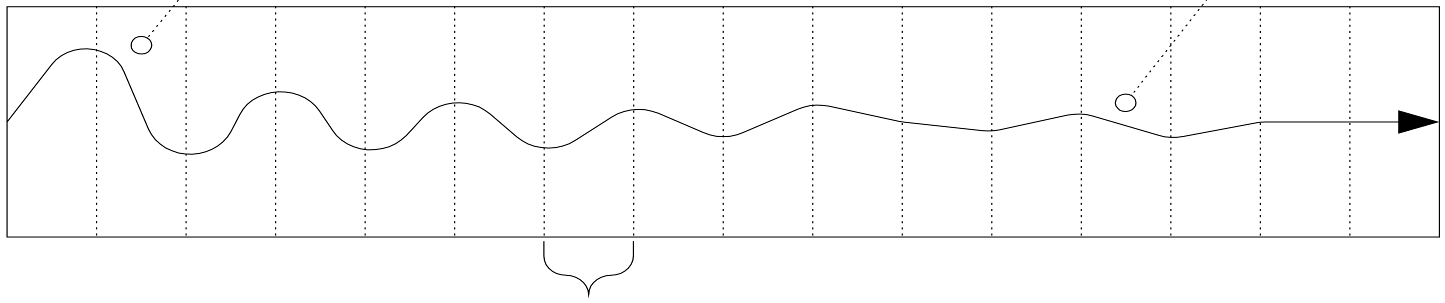

Requirements
Design
Implementation & Test & Integration & More Design
Final Integration & System Test
Time
Requirements
Design
Implementation & Test & Integration & More Design
Final Integration & System Test
Feedback from iteration N leads to refinement and adaptation of the requirements and design in iteration N+1.
3 weeks (for example)
Iterations are fixed in length, or *timeboxed*.
The system grows incrementally.

Convergence to true system path

- Build – Feedback – Adapt Cycles
- Helps resolve and prove the risky and critical design decisions early rather than late
- Over time the system converges over its true path.

Early iterations are farther from the "true path" of the system. Via feedback and adaptation, the system converges towards the most appropriate requirements and design.

In late iterations, a significant change requirements is rare, but can occur. S late changes may give an organizatio competitive business advantage.



one iteration of design, implement, integrate, and test

Benefits of Iterative Development

- Less project failure, better productivity, and lower defect rates.
- Early mitigation of high risks.
- Early visible progress.
- Early user engagement provides feedback for system adaptation to true stakeholder needs.
- Managed complexity: avoids analysis paralysis.
- A chance to improve the process itself iteration by iteration.

Iteration Timeboxing

- Iterations should be short.
  - Between two to three weeks.
  - Less than two weeks: too short for meaningful throughput and feedback.
  - More than six weeks: complexity becomes overwhelming
- Iterations have fixed length (timeboxing)
  - If it seems difficult meeting the deadline de-scope (move tasks or requirements to future iterations).

Why is Waterfall Bad?

- It is a proven fact that on average 45% of the features of waterfall projects was never used.
- 25% change in requirements is typical in software projects. Therefore is impossible to freeze the requirements early on.

Risk-Driven and Client-Driven Iterative Planning

- The goals of early iterations are:
  1. To identify and drive down the high risks
     - Early iterations focus on building, testing and stabilizing the core architecture.
  2. Build visible features that clients care most about.

Agile Processes

- It is not possible to exactly define **agile methods.** However main characteristics include short timeboxed iterations with evolutionary refinement of plans, requirements, and design.
- In addition, they promote practices and principles that reflect an agile sensibility of simplicity, lightness, communication, self-organizing teams, and more.

Critical Unified Process Practices

- Basic:
  - Short timeboxed iterations
  - Evolutionary and adaptive development
- In Addition:
  - Address high-risk and high-value early
  - Engage users
  - Build core architecture early
  - Continuously verify quality (test often, early and realistically)

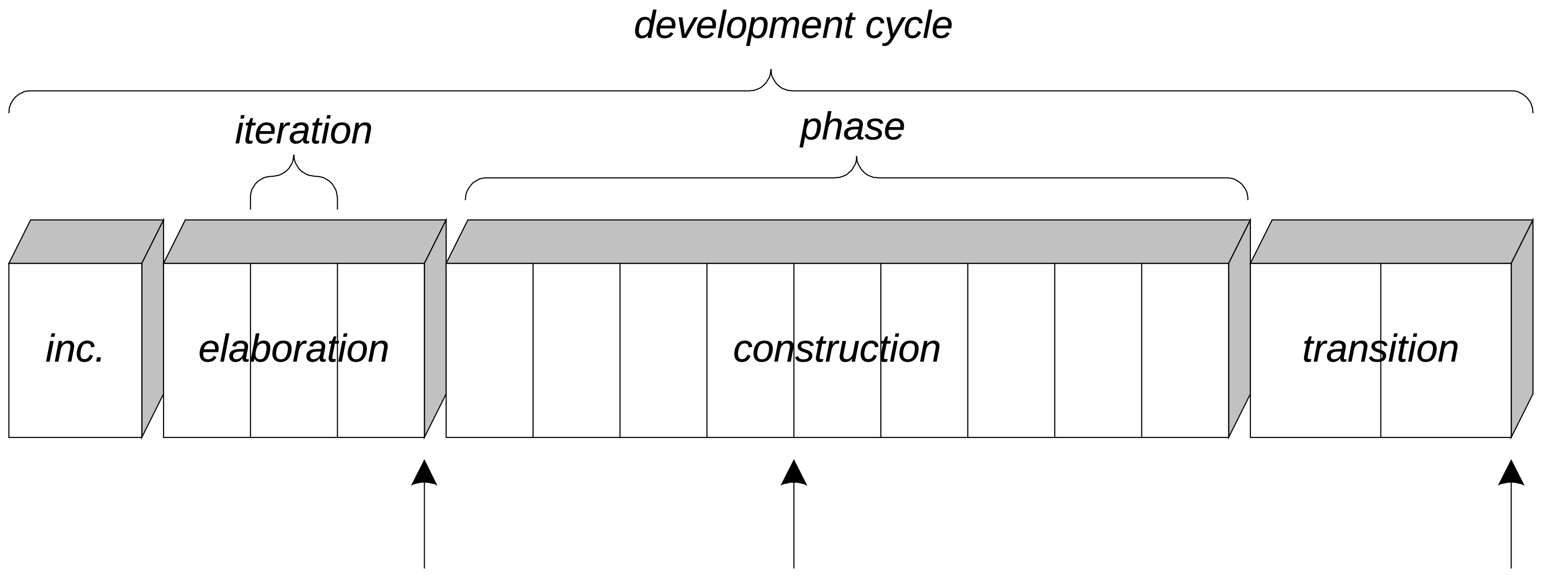
Critical Unified Process Practices (cont.)

- Apply use cases where appropriate
- Do some visual modeling (with UML)
- Carefully manage requirements
- Practice change requrest and configuration management.

Unified Process Phases

- Inception – approximate vision, business case, scope, vague cost estimates, buy and/or build.
- Elaboration – refined vision, iterative implementation of the core architecture, resolution of high risks, identification of most requirements and scope, more realistic estimates.
- Construction – iterative implementation of the remaining lower risk and easier elements, and preparation for deployment.
- Transition – beta tests, deployment.

Schedule-Oriented Terms in the Unified Process



milestone

An iteration end-point when some significant decision or evaluation occurs.

release

A stable executable subset of the final product. The end of each iteration is a minor release.

increment

The difference (delta) between the releases of 2 subsequent iterations.

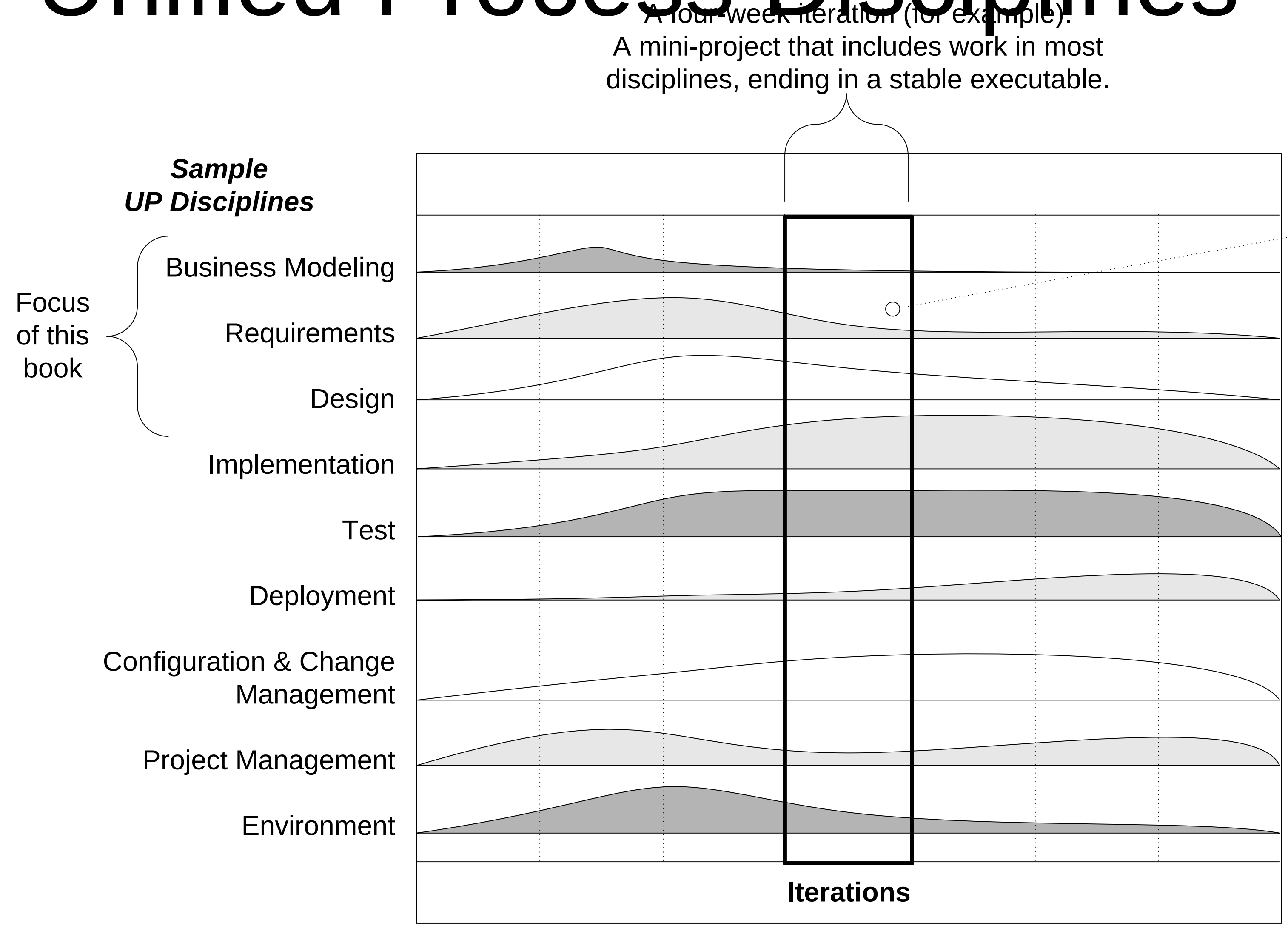
final production release

At this point, the system is released for production use.

Some Unified Process Terminology

- Artifact: A general term for any work product: code, web graphics, database schema, text documents, diagrams, models and so on.
- Discipline: A set of activities and related artifacts in one subject area such as the activities within requirements analysis.
- *The UP describes work activities, such as writing a use case, within disciplines.*

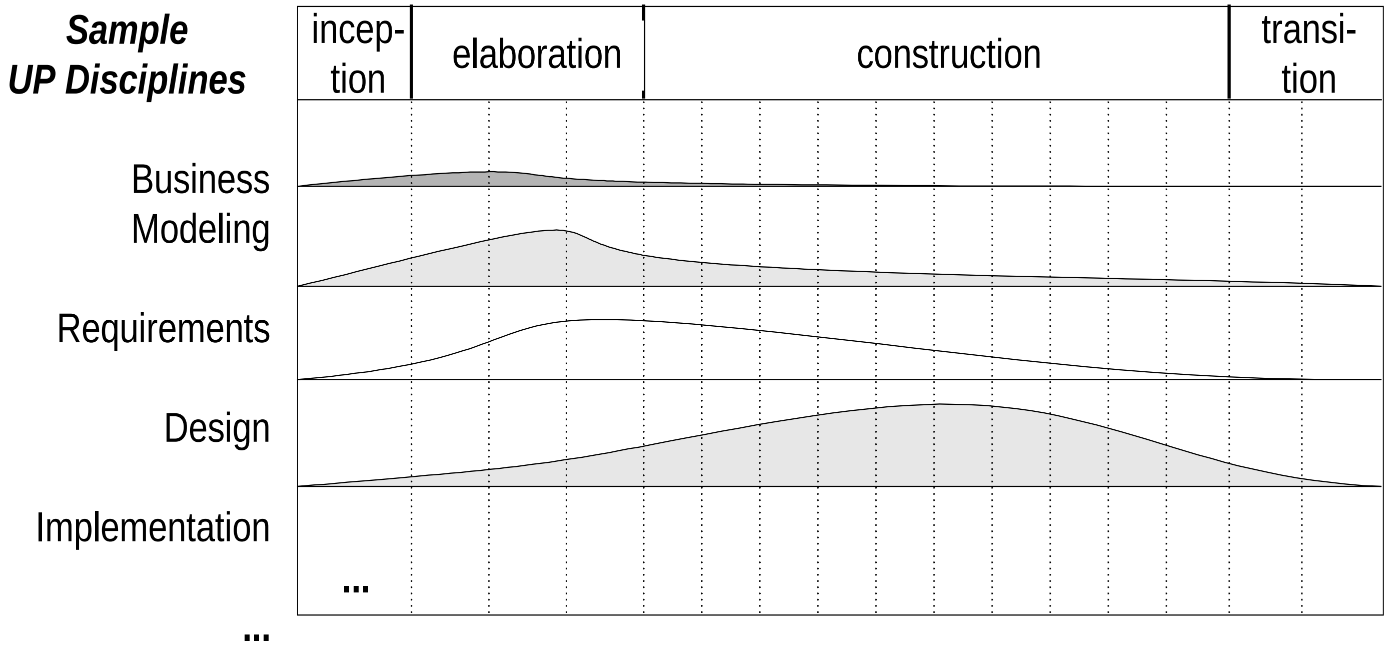
Unified Process Disciplines



Note that although an iteration includes work in most disciplines, the relative effort and emphasis change over time.

This example is suggestive, not literal.

Relationship Between Disciplines and Phases



The relative effort in disciplines shifts across the phases.

This example is suggestive, not literal.

Inception Phase

- Explores the following questions:
  - What is the vision and business case for this project?
  - Feasible?
  - Buy and/or build?
  - *Rough* unreliable range of cost: Is it $10K-100K or in the millions?
  - Should we proceed or stop?
- Should be short (e.g. one week for most projects)

Artifacts that **may start** in Inception

- Vision and business case
- Use-Case model
- Supplementary Specification
- Glossary
- Risk List & Risk Management Plan
- Prototypes and Proof-of-Concepts
- Iteration Plan
- Phase Plan & Software Development Plan
- Development Case

Requirements and Requirements Analysis

- “Requirements are capabilities and conditions to which the system – and more broadly, the project – must conform” [Jacobson, Booch and Rambaugh: “The Unified Software Development Process”, Addison-Wesley, 1999]
- A prime challenge of requirement analysis is to find, communicate and remember (write down) what is really needed, in a form that is clear both to clients and team members.

Types and Categories of Requirements

- In UP requirements are categorized according to the FURPS+ model [R. Grady: “Practical Software Metrics for Project Management and Process Improvement”, Prentice-Hall Inc, 1992.]
  - **F**unctional – features, capabilities, security
  - **U**sability – human factors, help, documentation
  - **R**eliability – frequency of failure, recoverability, predictability
  - **P**erformance: response times, throughput, accuracy, availability, resource usage
  - **S**upportability – adaptability, maintainability, internationalization, configurability

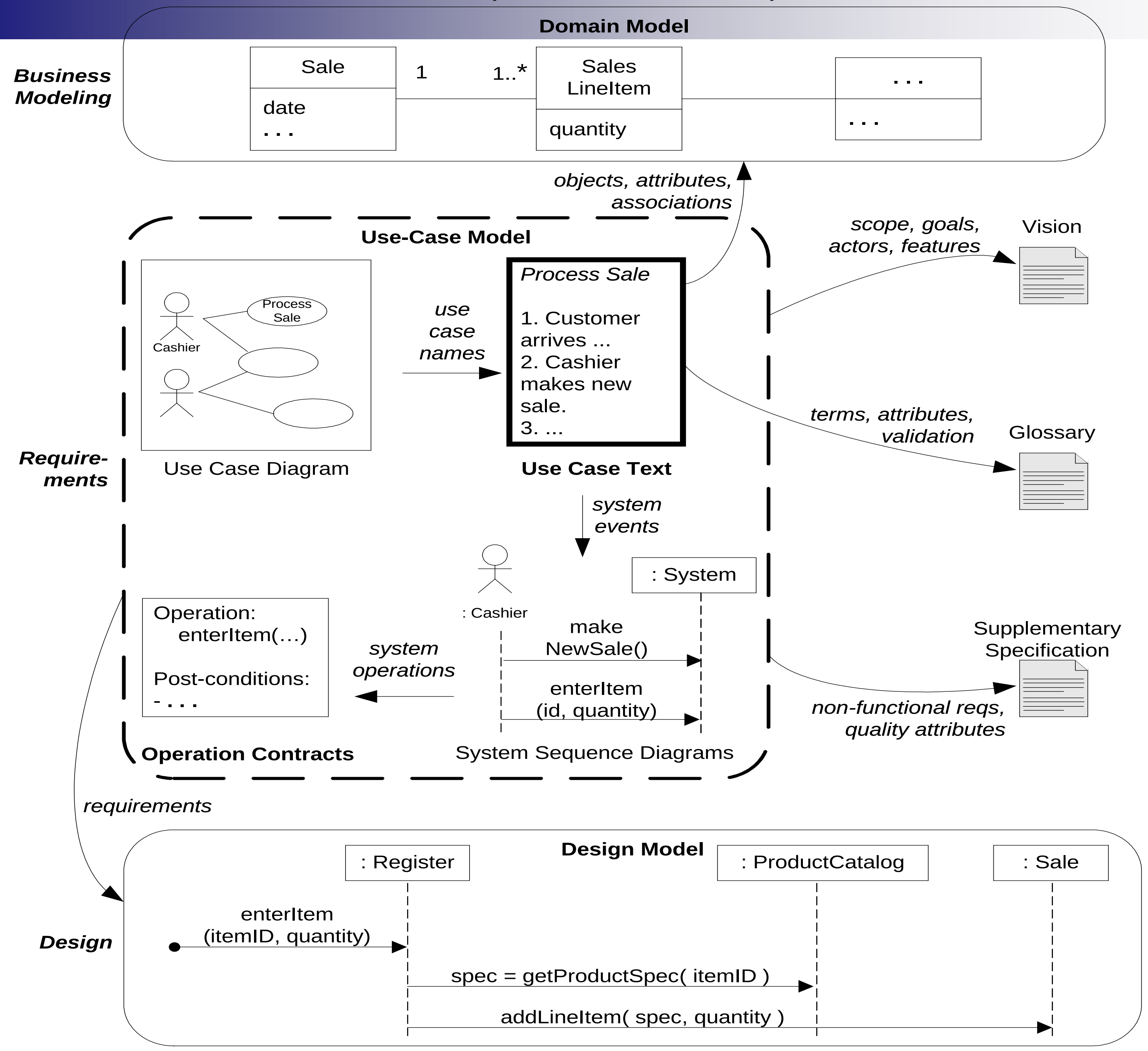
Types and Categories of Requirements (cont.)

- The ‘+’ in FURPS+ indicates sub-factors such as:
  - Implementation – resource limitations, languages and tools, hardware, …
  - Interface – constraints imposed by interfacing with external systems
  - Operations – system management in its operational setting
  - Packaging – for example a physical box
  - Legal – Licensing and so forth

Use Cases

- Informally they are stories of system usage.
- A mechanism to capture requirements
- They are written in a text form but can also be drawn in diagrams.
    - The text is more important: writing use cases is a text-writing activity: **Use cases are not diagrams, they are text.**
- The hard part is not learning the use case diagram, which is very simple, but learning how to identify and write good use cases.

Sample UP Artifact Relationships



Brief Format of Use Cases

- Uses cases are text describing the usage of the system by an actor to meet a goal.
- Initially they can be written in a brief format, e.g.

  "*Process sale: A customer arrives at a checkout with items to purchase. The Cashier uses the POS (Point Of Sales) system to record each purchased item. The system presents a running total and line-item details. The customer enters payment information, which the system validates and records. The system updates inventory. The customer receives a receipt from the system and leaves with the items.*"

Actors, Scenarios and Use Cases

- An actor is something with behavior, such as a person (identified by role), computer system, or organization, e.g. Cashier.
- A scenario is a specific sequence of actions and interactions between actors and the system (also called a use case instance).
- A use case is a collection of related *success and failure scenarios*, that describe an actor using the system to achieve a goal.

Use case definition by RUP

- Another use case definition by RUP: “A set of use-case instances, where each instance is a sequence of actions a system performs that yields an observable result of value to a particular actor”.
- The use case model may optionally include a use case diagram:
  - Names of use cases and actors
  - A context diagram of system and its environment
  - A quick way to list use cases by names

Why use cases?

- They are a simple way of capturing user goals: they don’t send your average business person to coma!
- Emphasize the user goals and perspective.
- Don’t concentrate on secondary issues (use case relationships, use case packages and so forth). Instead do the **hard work** of simply writing text stories.

Three Kinds of Actors

- **Primary Actor**: has user goals fulfilled through using services of the SuD (System under Development) – e.g. Cashier
  - Why identify? To find user goals, which drive use cases.
- **Supporting Actor:** provides a service to the SuD (e.g. payment authorization service).
  - Why identify? Clarify external interfaces and protocols
- **Offstage Actor:** has an interest in the behavior of the use case (but is not primary or supporting) – e.g. tax agency.
  - Why identify? To ensure than all necessary interests are identified and satisfied.

Three Common Use Case Formats

- **Brief:** One paragraph summary (usually the main success scenario *or* happy path)
  - When? During early requirements analysis.
- **Casual:** Multiple paragraphs covering various scenarios.
  - When? As above
- **Fully Dressed:** Includes all steps and variations. Also includes supporting sections (e.g. preconditions)
  - When? After many use cases have been identified and written in brief then in the 1st requirements workshop 10% of them (<u>the most important</u>) are written in detail.

Use Case Template

- The most widely used template for **fully dressed** use cases is the one suggested by Alistair Cockburn (http://alistair.cockburn.us/

Use Case Section	Comment
Use Case Name	Start with a verb
Scope	The system under design
Level	“user goal” or “subfunction”
Primary Actor	Calls on the system to deliver its services
Stakeholders and Interests	Who cares about this use case and what do they want.

Use Case Template (cont.)

Use Case Section	Comment
Preconditions	What must be true on start and worth telling the reader
Success Guarantees	What must be true on succesful completion, and worth telling the reader
Main Success Scenario	A typical, unconditional, happy path scenario of success
Extensions	Alternate scenarios of success and failure

Use Case Template (cont.)

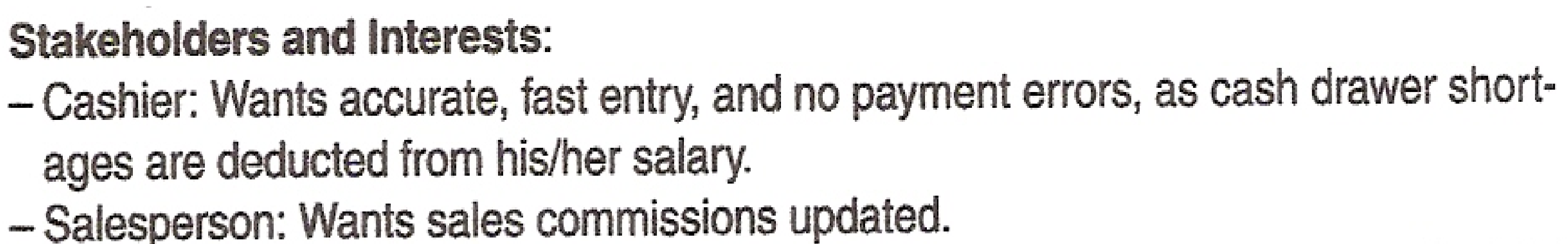
Use Case Section	Comment
Special Requirements	Related non-functional requirements
Technology and data variations list	Varying I/O methods and data formats
Frequency of occurrence	Influences investigation, testing and timing of implementation
Miscellaneous	Such as open issues

Scope and Level

- Scope:
  - System use case: describes the use of one software (or hardware+software) system
  - Business use case: enterprise-level process description
- Level:
  - User-Goal level: a common kind that describes the scenarios to fulfill the goals of a primary actor (corresponds to an Elementary Business Process – EBP)
  - Subfunction level: substeps required to support a user goal. Factors out duplicate substeps in many use cases (e.g. *Pay by Credit)*.

Primary Actors – Stakeholders and Interests List

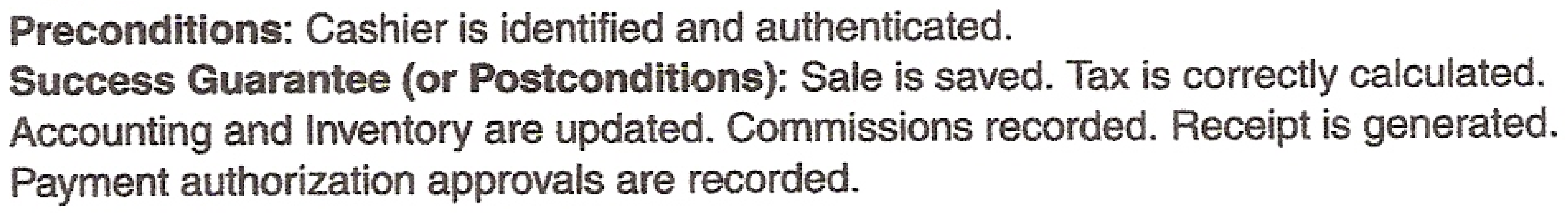
- **Primary Actor**: the principal actor that calls upon system services to fulfill a goal.
- **Stakeholders and Interests** list: Very important since identifying all stakeholders and their interests answers the question ‘What should be in the use case?’ – ‘That which satisfies all stakeholders’ interests’



Stakeholders and Interests:
- Cashier: Wants accurate, fast entry, and no payment errors, as cash drawer shortages are deducted from his/her salary.
- Salesperson: Wants sales commissions updated.

Preconditions – Success Guarantees (Postconditions)

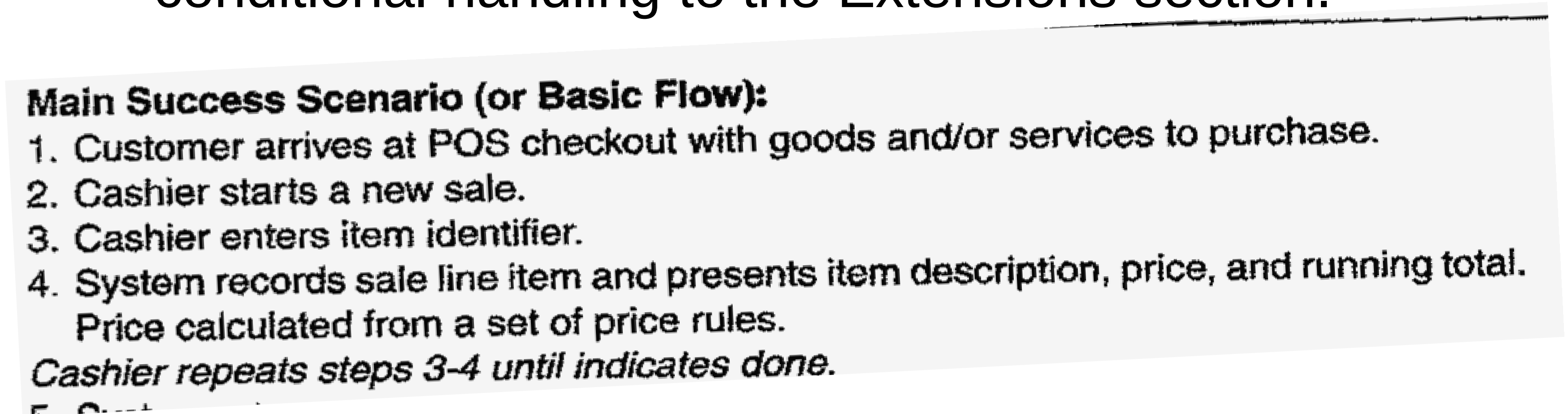
- Mention them only when you are stating something non-obvious and noteworthy
- Preconditions: state what must always be true **before** a scenario is began.
  - Imply completion of another scenario (e.g. login)
  - Don’t mention them if obvious (e.g. the system has power)
- Success guarantees: state what must be true on successful completion of the use case. Should meet the needs of all stakeholders.



Preconditions: Cashier is identified and authenticated.
Success Guarantee (or Postconditions): Sale is saved. Tax is correctly calculated. Accounting and Inventory are updated. Commissions recorded. Receipt is generated. Payment authorization approvals are recorded.

Main Success Scenario (or Basic Flow)

- Also called the ‘happy path’ scenario. Describes the typical success path that satisfies the interests of the stakeholders.
- Often does not include any conditions of branching.
  - More comprehensible and extensible to defer all conditional handling to the Extensions section.



Main Success Scenario (or Basic Flow):

1. Customer arrives at POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. System records sale line item and presents item description, price, and running total. Price calculated from a set of price rules.

Cashier repeats steps 3-4 until indicates done.

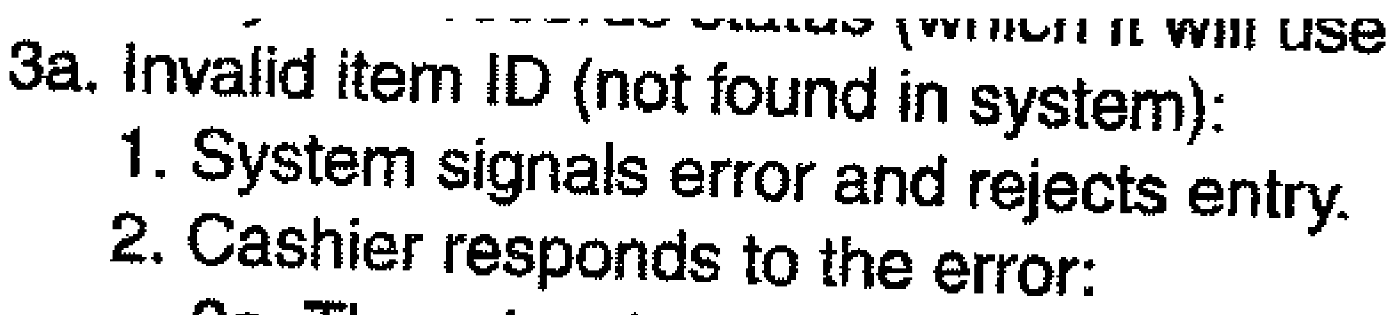
Three Kinds of Steps in scenarios

1. An interaction between actors
   - Not that SuD itself should be considered as an actor when it plays an actor role collaborating with other systems.
2. A validation (usually by the system)
3. A state change by the system (for example recording or modifying something).

   The first step is a **trigger** that starts the scenario

Extensions or Alternate Flows

- Extensions comprise the majority of the text, indicating all other scenarios (besides the main success scenario) success and failures.
- Main success scenario + extensions: Should satisfy "nearly" all stakeholders' interests.
  - Non-functional requirements are expressed in the *Supplementary Specification*
- Extensions are branches from the main success scenario with respect to its steps and have two parts: (a) the condition and (b) its handling.



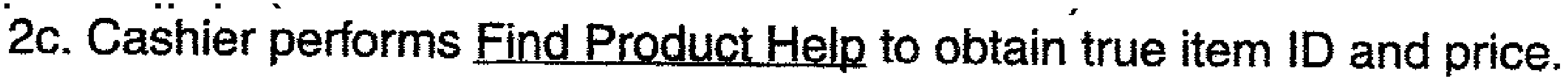
status (which it will use

3a. Invalid item ID (not found in system):

1. System signals error and rejects entry.
2. Cashier responds to the error:

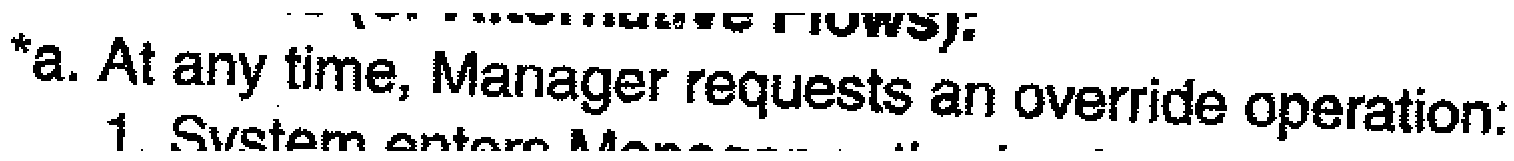
Extensions (cont.)

- Extension handling can be summarized in one step or include a sequence of steps.
- At the end of the extension handling, by default the scenario merges back with the main success scenario, unless the extension indicates otherwise (e.g. the system halts)
- Complicated extension points (e.g. paying by credit) are usually expressed as separate use cases (with



2c. Cashier performs Find Product Help to obtain true item ID and price.

- If an extension is possible during any step then is marked with an asterisk.

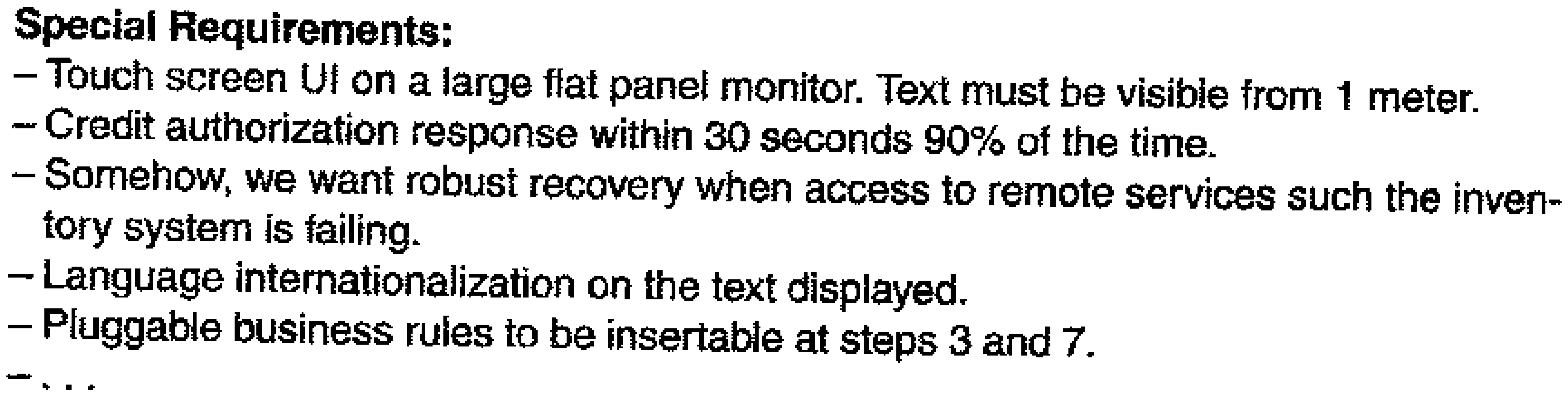


*a. At any time, Manager requests an override operation:

1. System

Special Requirements

- Non-functional requirements that relate to the use case can be mentioned in the *Special Requirements* sections.
- Many prefer mentioning all non-functional requirements in the *Supplementary Specification* artifact, because usually all non-functional requirements are considered as a whole during architectural analysis.

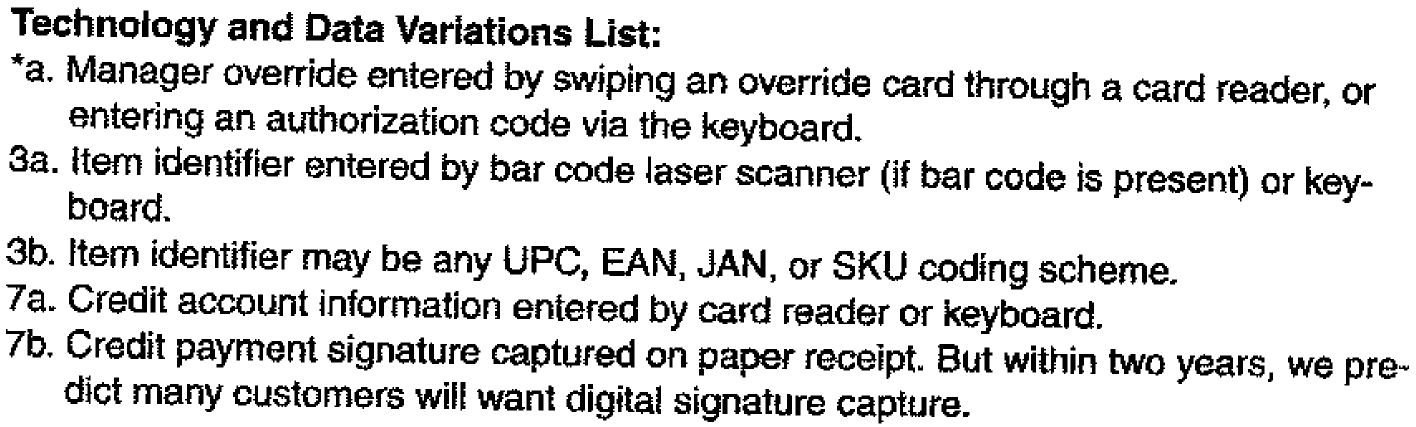


Special Requirements:

– Touch screen UI on a large flat panel monitor. Text must be visible from 1 meter.
– Credit authorization response within 30 seconds 90% of the time.
– Somehow, we want robust recovery when access to remote services such the inventory system is failing.
– Language internationalization on the text displayed.
– Pluggable business rules to be insertable at steps 3 and 7.
– . . .

Technology and Data Variations List

- Technical variations in *how* something must be done.
  - Example: Constraints imposed by stakeholders for input/output devices (e.g. POS system must support credit account input both using a card reader and keyboard"
  - It is skillful to avoid these early commitments however this is not always possible.
- Variations in data schemes are also sometimes necessary, such as UPCs or EANs for item identifiers



Technology and Data Variations List:

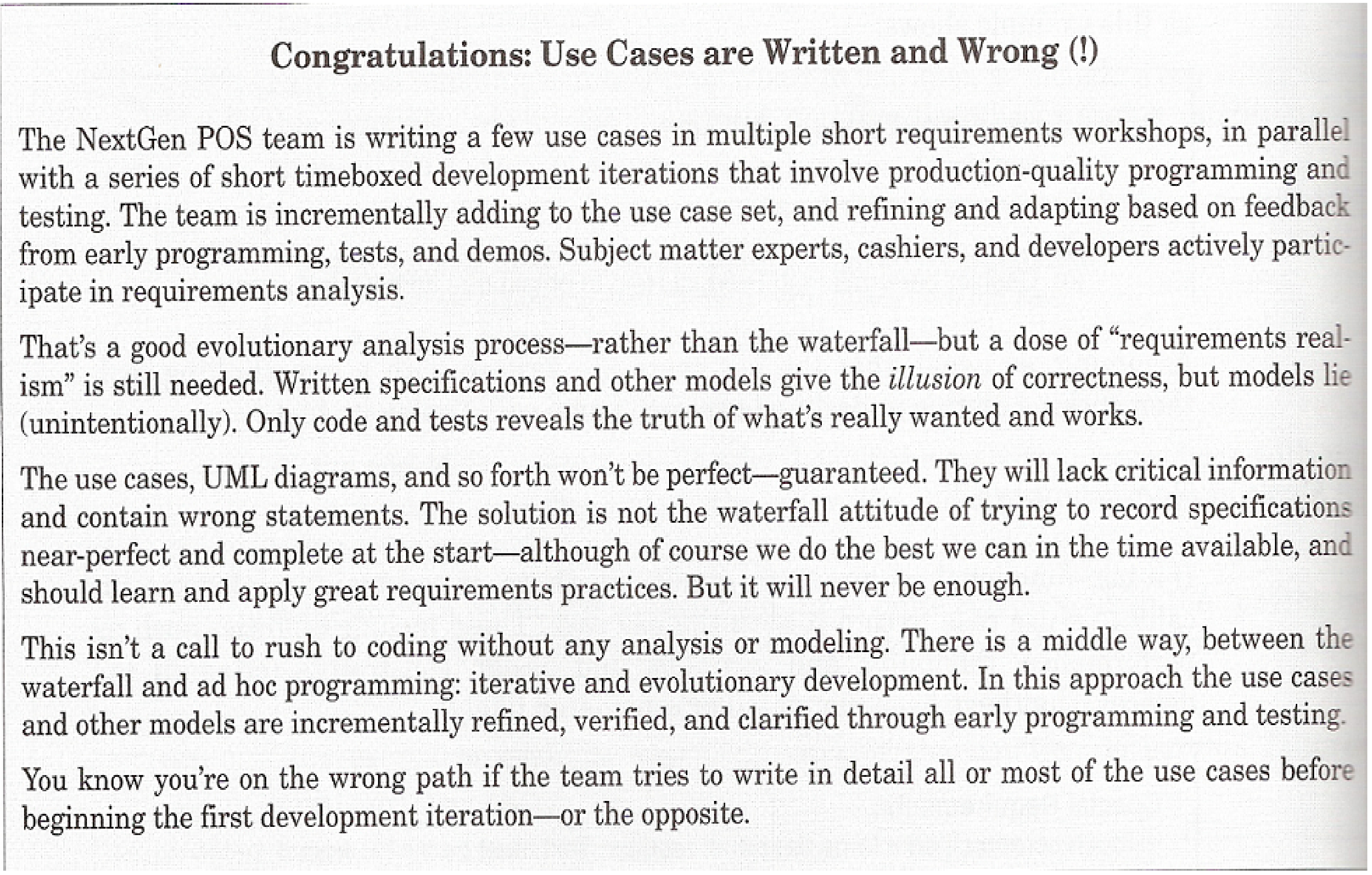
*a. Manager override entered by swiping an override card through a card reader, or entering an authorization code via the keyboard.

3a. Item identifier entered by bar code laser scanner (if bar code is present) or keyboard.

3b. Item identifier may be any UPC, EAN, JAN, or SKU coding scheme.

7a. Credit account information entered by card reader or keyboard.

7b. Credit payment signature captured on paper receipt. But within two years, we predict many customers will want digital signature capture.



Congratulations: Use Cases are Written and Wrong (!)

The NextGen POS team is writing a few use cases in multiple short requirements workshops, in parallel with a series of short timeboxed development iterations that involve production-quality programming and testing. The team is incrementally adding to the use case set, and refining and adapting based on feedback from early programming, tests, and demos. Subject matter experts, cashiers, and developers actively participate in requirements analysis.

That's a good evolutionary analysis process—rather than the waterfall—but a dose of "requirements realism" is still needed. Written specifications and other models give the *illusion* of correctness, but models lie (unintentionally). Only code and tests reveals the truth of what's really wanted and works.

The use cases, UML diagrams, and so forth won't be perfect—guaranteed. They will lack critical information and contain wrong statements. The solution is not the waterfall attitude of trying to record specifications near-perfect and complete at the start—although of course we do the best we can in the time available, and should learn and apply great requirements practices. But it will never be enough.

This isn't a call to rush to coding without any analysis or modeling. There is a middle way, between the waterfall and ad hoc programming: iterative and evolutionary development. In this approach the use cases and other models are incrementally refined, verified, and clarified through early programming and testing.

You know you're on the wrong path if the team tries to write in detail all or most of the use cases before beginning the first development iteration—or the opposite.

Guideline 1: Prefer Essential UI-Free Use Cases

- Ask *what is the goal of that goal?*
  - Example:
    - Cashier wants to login => login screen
      - Identify myself and get authenticated => any authentication method will do (e.g. biometric reader on keyboard)
        - Prevent theaft
    - If the true goal is authentication then why not make it fast and easy (e.g. fingerprinting) => usability analysis (i.e. are their fingers covered in grease? Do they have fingers?)
- Essential writing style is expressing user intentions and system responsibilities, rather than concrete actions.

Guideline: Write Use Cases in an essential style; keep the user interface out and focus on actor intent.

Guideline 1: Prefer Essential UI-Free Use Cases (cont.)

- Essential style
    1. Administrator identifies itself
    2. System authenticates
- Concrete style
    1. Administrator enters id and password in dialog box (see picture 3)
- Concrete use cases may be useful during GUI design in a later phase, but are better avoided during early requirements analysis.

Guideline 2: Write terse use cases

- Keep your use cases short and to the point.
- Avoid noise words

Guideline: Write terse use cases.

Guideline 3: Write Black-Box Use Cases

- Don’t describe the internal working of the system, its components or design.
- Concentrate on responsibilities
  - Define *what* the system does (analysis), rather than
  - *How* it does it (design)

Guideline: Write black-box use cases.

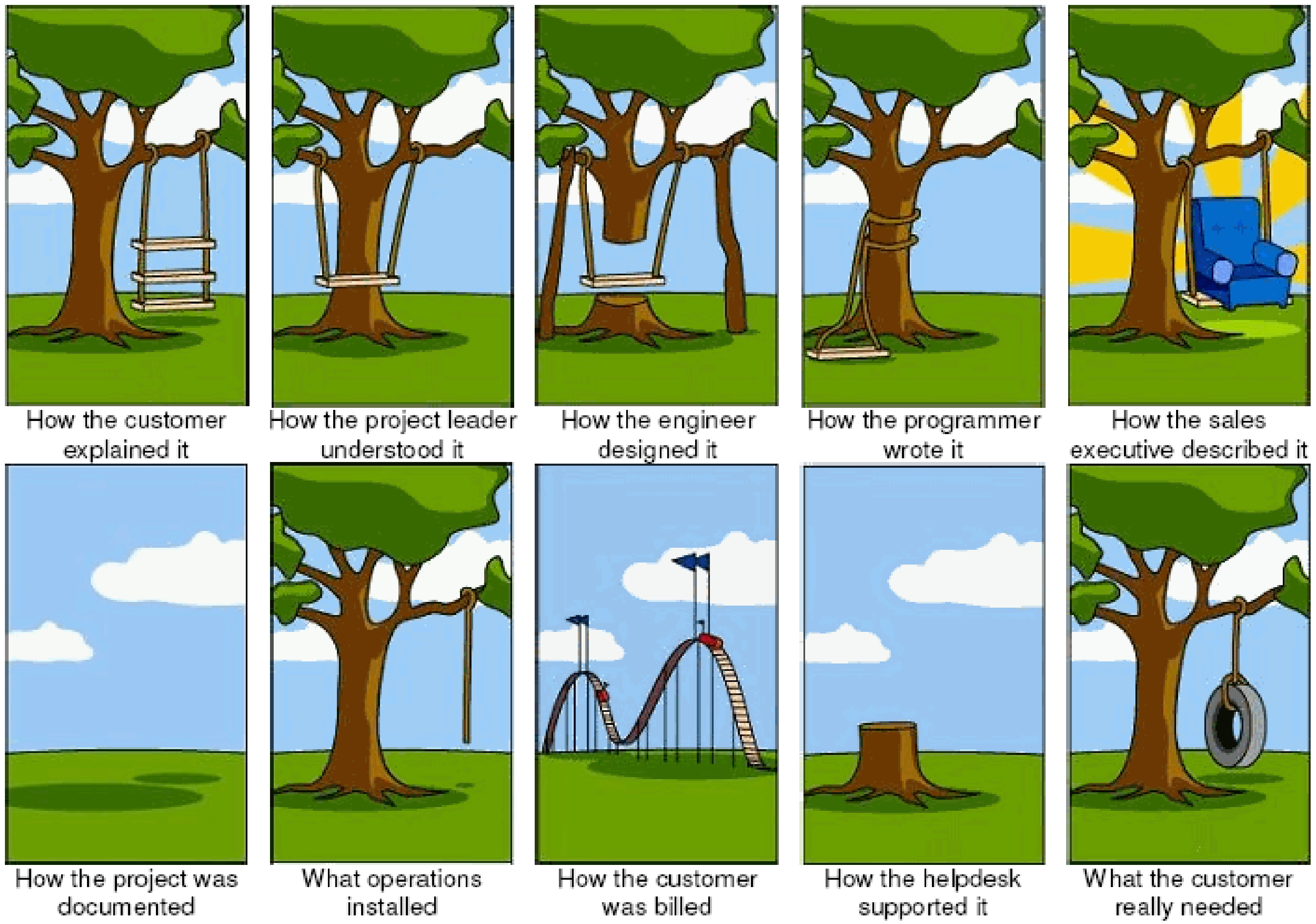
Black-box style	Not Black-box style
The system records the sale.	The system writes the sale to a database … or (even worse); The system generates a SQL INSERT statement for the sale

Guideline 4: Take an Actor and Goal Perspective

- Write requirements that focus on users or actors of a system, asking what goals they try to satisfy ("an observable result of value to a particular actor")
- Focus on understanding what the actors considers as valuable result.

Guideline: Take an Actor and Goal Perspective

User goals are important!


How the customer explained it
How the project leader understood it
How the engineer designed it
How the programmer wrote it
How the sales executive described it
How the project was documented
What operations installed
How the customer was billed
How the helpdesk supported it
What the customer really needed

How to Find Use Cases

- Choose the system boundary
  - Just the software application?
  - Software + Hardware?
  - Software + Hardware + Person?
  - Organization?

- Identify the primary actors
  - These questions might help: (a) Who starts and stops the system? (b) Who does user and security management? (c) Who does system administration? (d) Is time an actor (e.g. real-time systems) (e) Is there a monitoring process that restarts the system if it fails? (f) Push or pull updates? (g) Are there any external or robotic systems involved? (h) Who evaluates system activity or performance? (i) Who evaluates logs? Are they remotely retrieved? (j) Who gets notified when there are errors or failures?

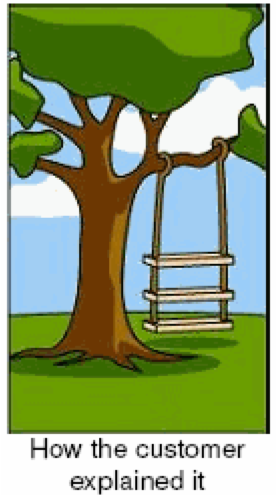
How to Find Use Cases (cont.)

- Identify the goals for each primary actor
  - Actors and Goals are usually discovered together.
  - Start with actors
  - Find their goals, which may reveal more actors, and so on.
- Define use cases that satisfy user goals
  - As you discover goals, you can name your use cases, or
  - You can start with an Actor-Goal list and then name the use cases.

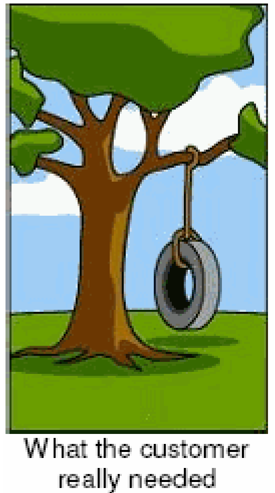
Why ask about Actor Goals rather than Use Cases?

- Asking about goals helps discovering real user requirements instead of current practices (and the complications that come with them)
- Remember the difference:

This is what you get by asking users about use cases

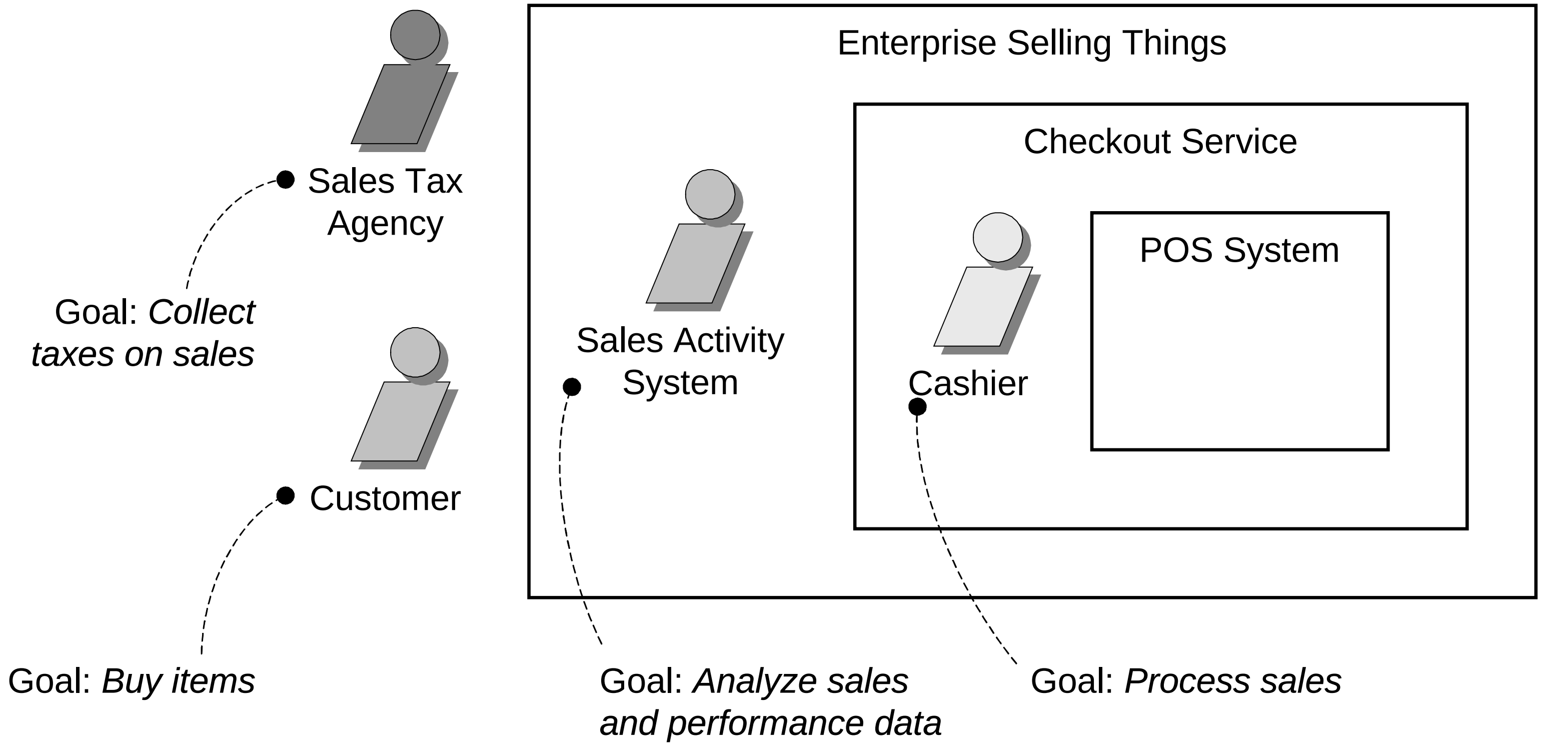


This is what you get by asking users about their goals



Who are the Primary Actors?

- It depends on the context!



Tests to Help You Find Use Cases

- Use cases can be at different levels:
  - Negotiate a supplier contract
  - Handle returns
  - Log in
  - Move Piece on Game Board

  An argument can be made that all the above are valid use case at different levels.
- What is a useful level for application requirements analysis? There are several rules of thumb:
  - The Boss Test
  - The EBP Test
  - The Size Test

The Boss Test

- Your boss asks, “What have you been doing all day?” You reply: “Logging in!” Is your boss happy?
- In this case probably not! A use case should have some real measurable value. Something that would make the boss happy.
- That said use cases that fail the boss test shouldn’t always be ignored: they might be low level, but important and difficult (such as user authentication in some cases).

The EBP Test

- What’s an Elementary Business Process (EBP)?
  - *A task performed by one person in one place at one time, in response to a business event, which adds measurable business value and leaves the data in a consistent state, e.g. Approve Credit or Price Order.*
- Don’t take the definition too literally
- EBPs are not single step actions (e.g. print a document), rather the main success scenario would be 5 or 10 steps.
- EBPs don’t require days to complete or multiple sessions.
- EBPs add observable ore measurable bussines value.
- For all the above are good candidates for use cases.

The Size Test

- A good use case shouldn’t be too short.
- Usually fully dressed use case descriptions require between 3 to 10 pages of text.

Candidate Use Cases Revisited

- Negotiate a supplier contract:
  - Much longer and broader than EBP (business use case, not a system use case)
- Handle returns
  - OK with the boss! Seems like an EBP. Size is good.
- Log in
  - Fails the boss test! Boss won’t be happy if that’s what you do all day long!
- Move Piece on Game Board
  - Single step – fails the size test.

Reasonable Violations of the Tests

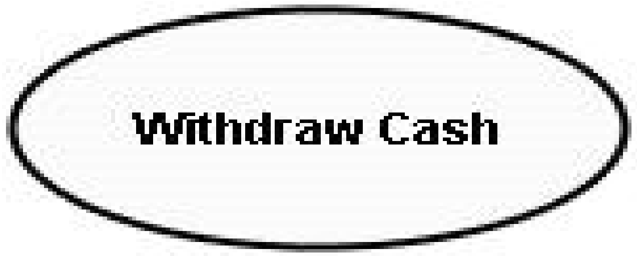
- It is sometimes useful to separate subfunctions as separate use cases, simply because they are included in many other use cases (e.g. *Pay by Credit*)
- Also *Authenticate User* may not pass the boss test, but be complex enough to warrant careful analysis, such as for a ‘single sign-on’ feature.

Use Case UML Diagrams

- Cockburn, Fowler, Larman and others downplay the importance of use case diagrams and all sugest using the text form instead.
- That said, use case diagrams can provide a nice summary of use cases, and the ways the actors use it.
- In the following slides we will show the basic UML diagrammatic elements for use cases, with the advice to keep it simple and concentrate in the writing of text use cases.

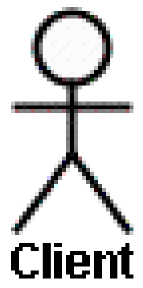
Use Case

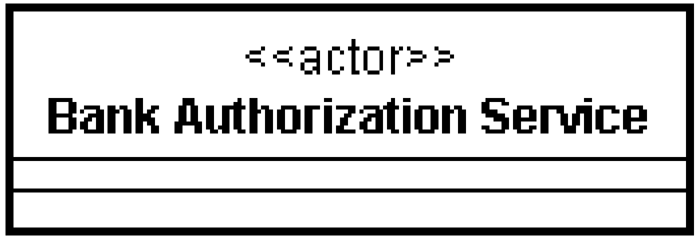
- Represents a user goal for an actor of the system
- Actors may be humans (e.g. secretary, cashier etc.), but also external systems (e.g. payment authorization service)
- The UML symbol for a use case is an ellipse with the name of the use case



Actors

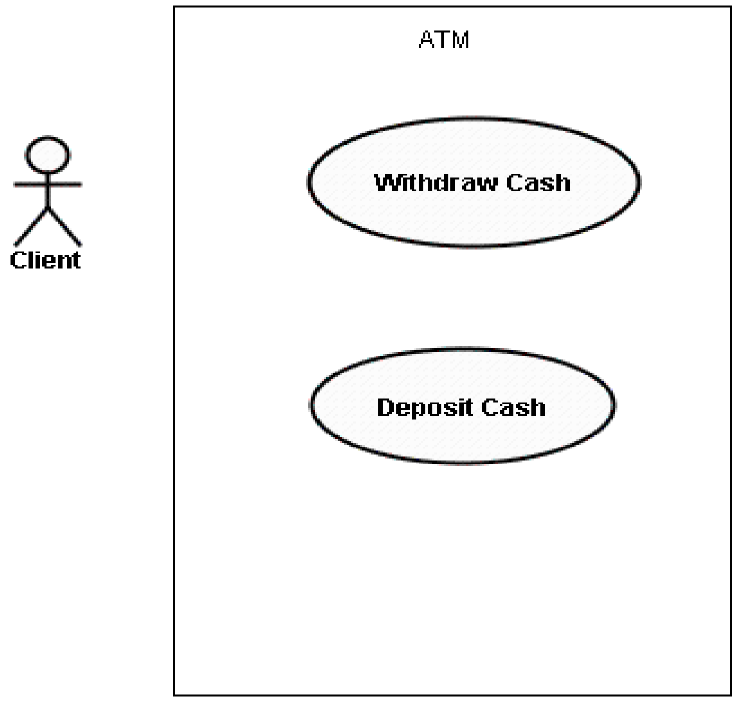
- Actors can be humans or subsystems
- The symbol of an actor is a stickman
- If the actor is a subsystem we suggest using an alternate symbol with the stereotype <<Actor>> for emphasis.





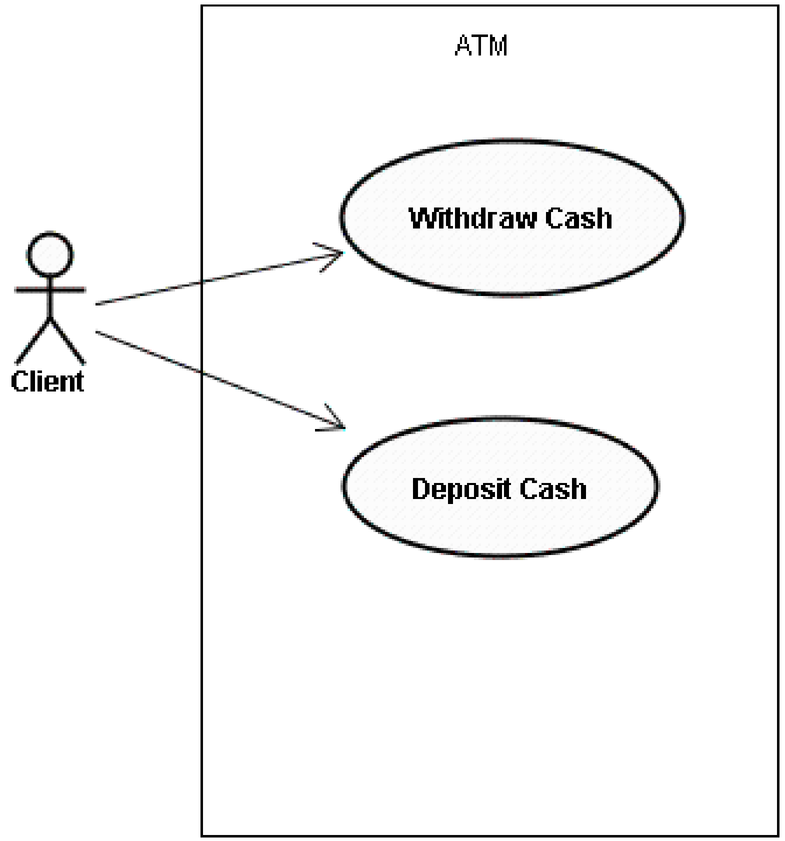
System Boundary

- To separate the requirements that are part of the SuD from external subsystems, we draw system use cases in a system box which visualizes the system boundary
- Inside the system box we place the use cases and outside it actors and external systems.



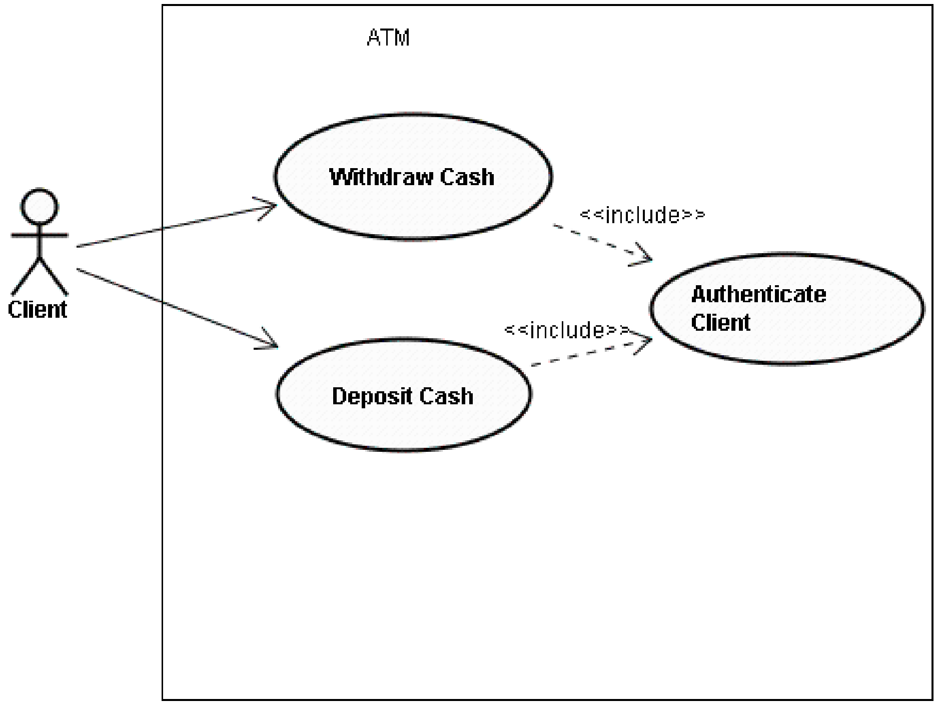
Association

- To declares the relationship between an actor and the SuD we associate the actor with the system use cases by drawing a line between the actor and the use case.
- Actors may be primary, supporting or offstage.



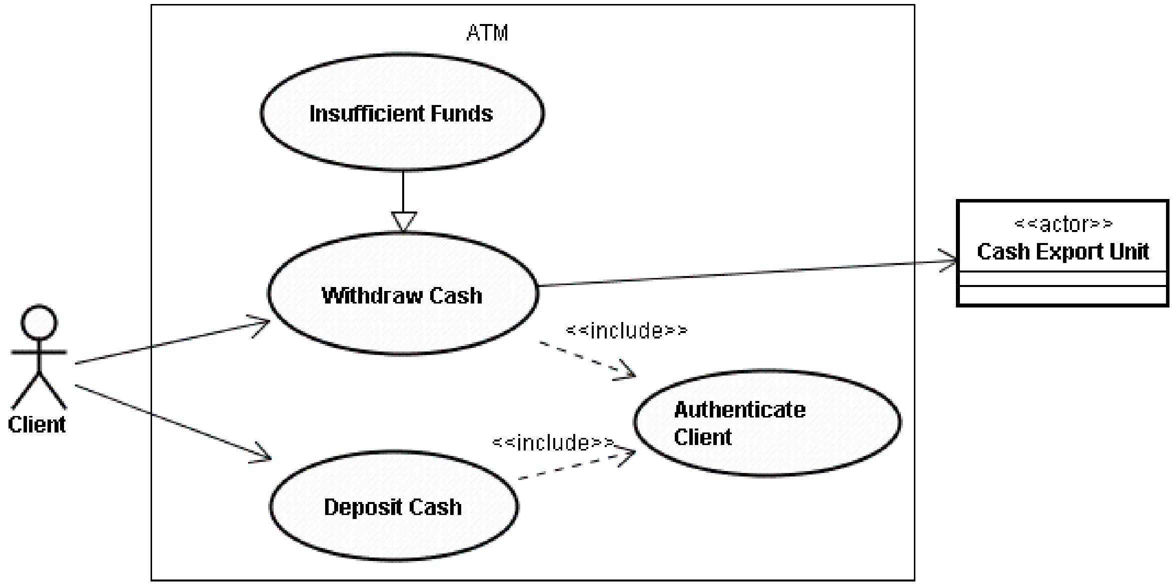
Include relationships

- Inclusion is a special type of association in which a use case always includes another use case.
- The direction of the arrow is from the use case that includes to the use case that is included.
- Include relationships are depicted with a dashed arrow



Extension

- Extension, similar to inclusion, is a relationship between two use cases, one which extends and the other who's extended.
- The extended use case is conditionally triggered by some condition (e.g. insufficient funds)


ATM
Insufficient Funds
Withdraw Cash
<<actor>>
Cash Export Unit
<<include>>
Client
Authenticate Client
<<include>>
Deposit Cash