Development Methodologies

Dr. James A. Bednar

jbednar@inf.ed.ac.uk

http://homepages.inf.ed.ac.uk/jbednar

Dr. David Robertson

dr@inf.ed.ac.uk

http://www.inf.ed.ac.uk/ssp/members/dave.htm

Development Methodologies

A methodology is a system of methods and principles used in a particular “school” of software design.

There is a wide variety of published methodologies, and an even larger set of informal and/or company-specific methodologies. The most mature methodologies are often codified using specialist tools and techniques.

All methodologies are controversial, because some people argue that any fixed methodology is an affront to a professional, creative, independent designer, while the rest argue about which methodology is best.

Example Methodologies

In this course we will discuss three main methodologies, and some variants:

- The Waterfall Model
- The Unified Process (UP)
- Extreme Programming (XP)

We will also discuss open-source design, which is more of a philosophical approach than a methodology like the others, but which has implications for methodology.

Waterfall Model

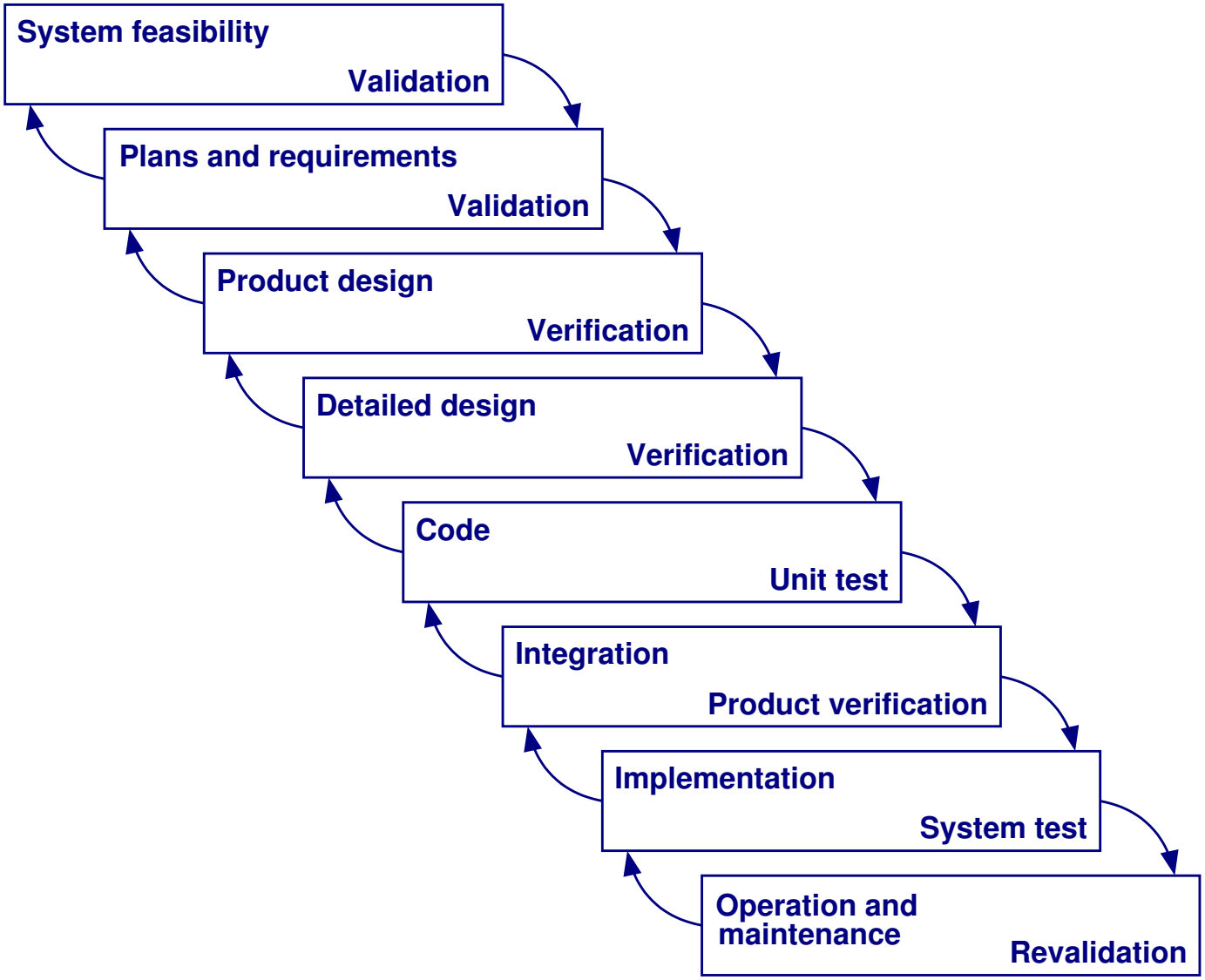
(Royce 1970) Inspired by older engineering disciplines, such as civil and mechanical (e.g. how cathedrals are built)

Development of a release is broken into phases, each of which is completed and “signed-off” before moving on.

When problems are found, must backtrack to a previous phase and start again with the sign-off procedures.

Much time and effort is spent on getting early phases right, because all later phases depend on them.

Waterfall Model of One Release



Problems with Waterfall Model

In practice it is rarely possible to go straight through from requirements to design to implementation, without backtracking.

There is no feedback on how well the system works, and how well it solves users' needs, until nearly the very end.

Large danger of catastrophic failure:

- Any error in key user requirements dooms entire process
- Big chance that the design is not actually feasible
- Big potential for unacceptable performance

The Unified Process

Modification of waterfall model to use modeling to forestall backtracking, add focus on OO, etc.:

- Component based
- Uses UML for all for all blueprints
- Use-case driven
- Architecture centric
- Iterative and incremental

Details in Jacobson et al. (1998).

Relatives of The Unified Process

The IBM Rational Unified Process (RUP) is a commercial product and toolset, superseding:

- The Objectory Process
- The Booch Method
- The Object Modeling Technique

The Unified Software Development Process (UP) is a published, non-proprietary method based on the RUP, but without specific commercial tools or proprietary methods.

Phases of UP Design

Each software release cycle proceeds through a series of phases, each of which can have multiple modeling iterations:

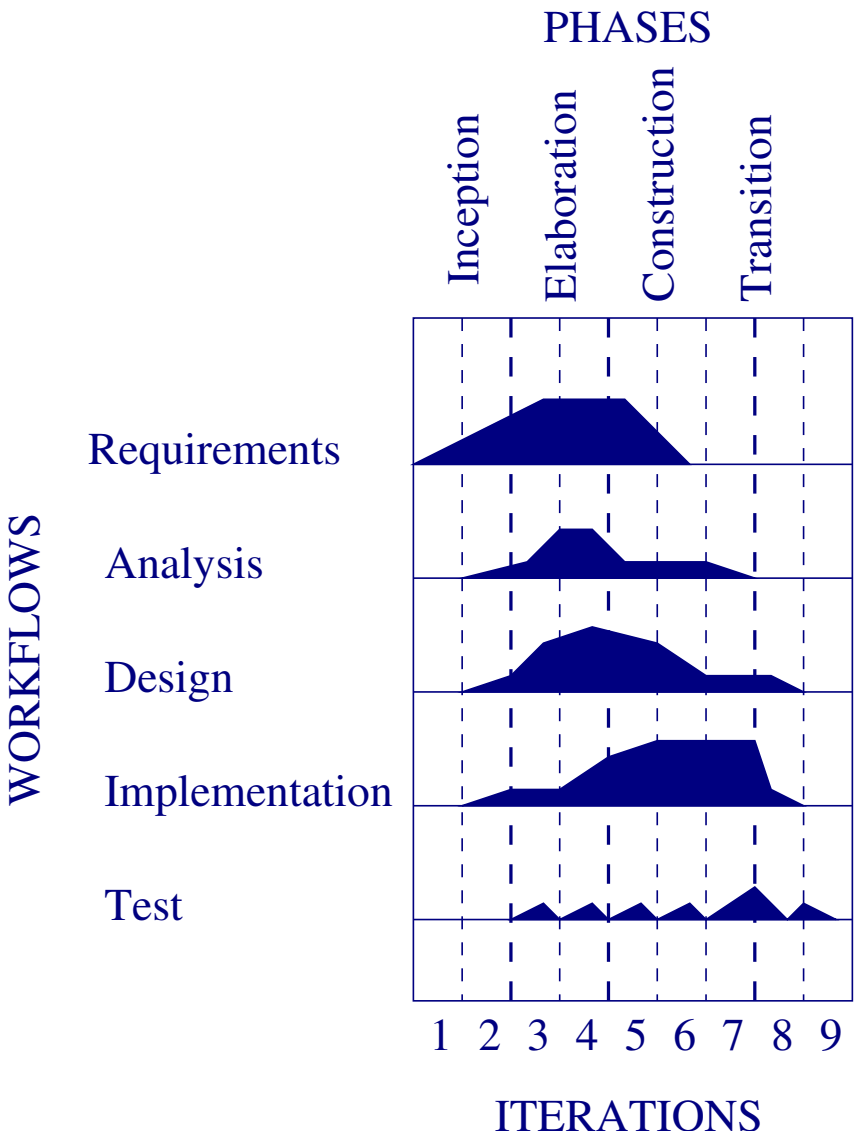
Inception : Produces commitment to go ahead
(business case feasibility and scope known)

Elaboration : Produces basic architecture;
plan of construction; significant risks identified;
major risks addressed

Construction : Produces beta-release system

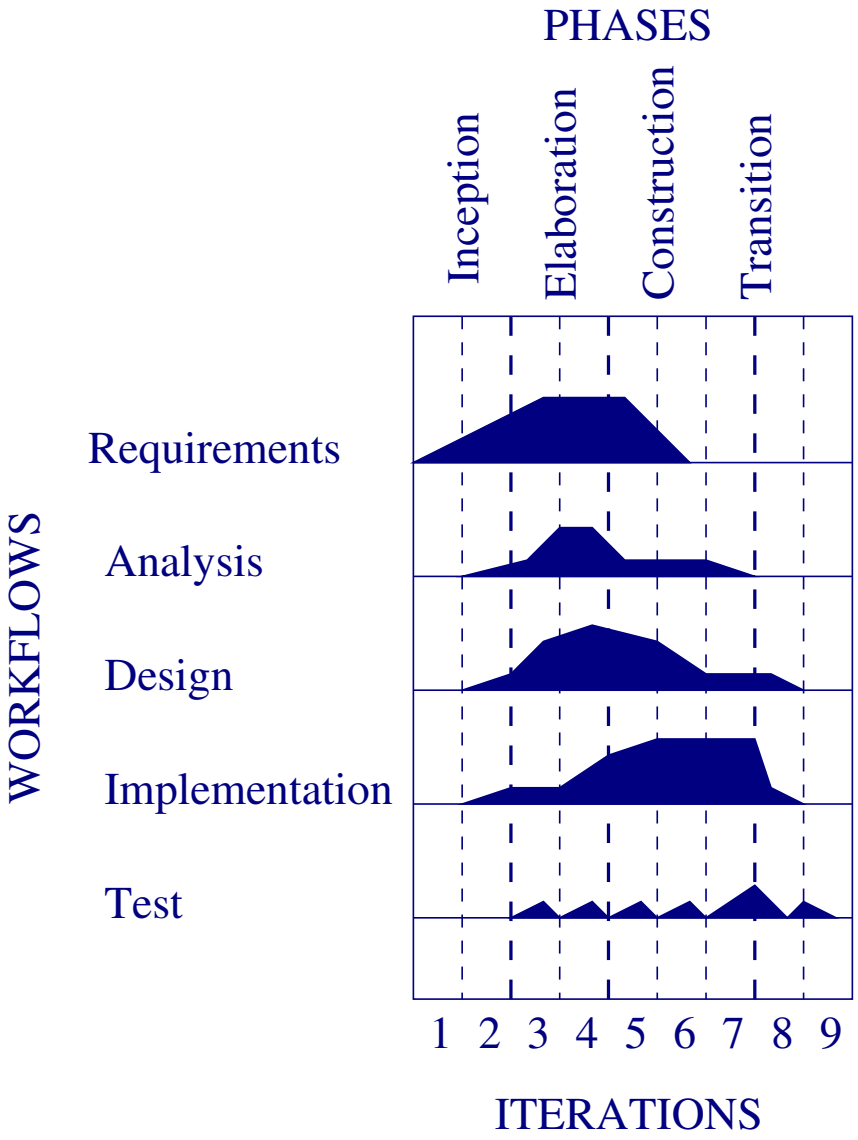
Transition : Introduces system to users

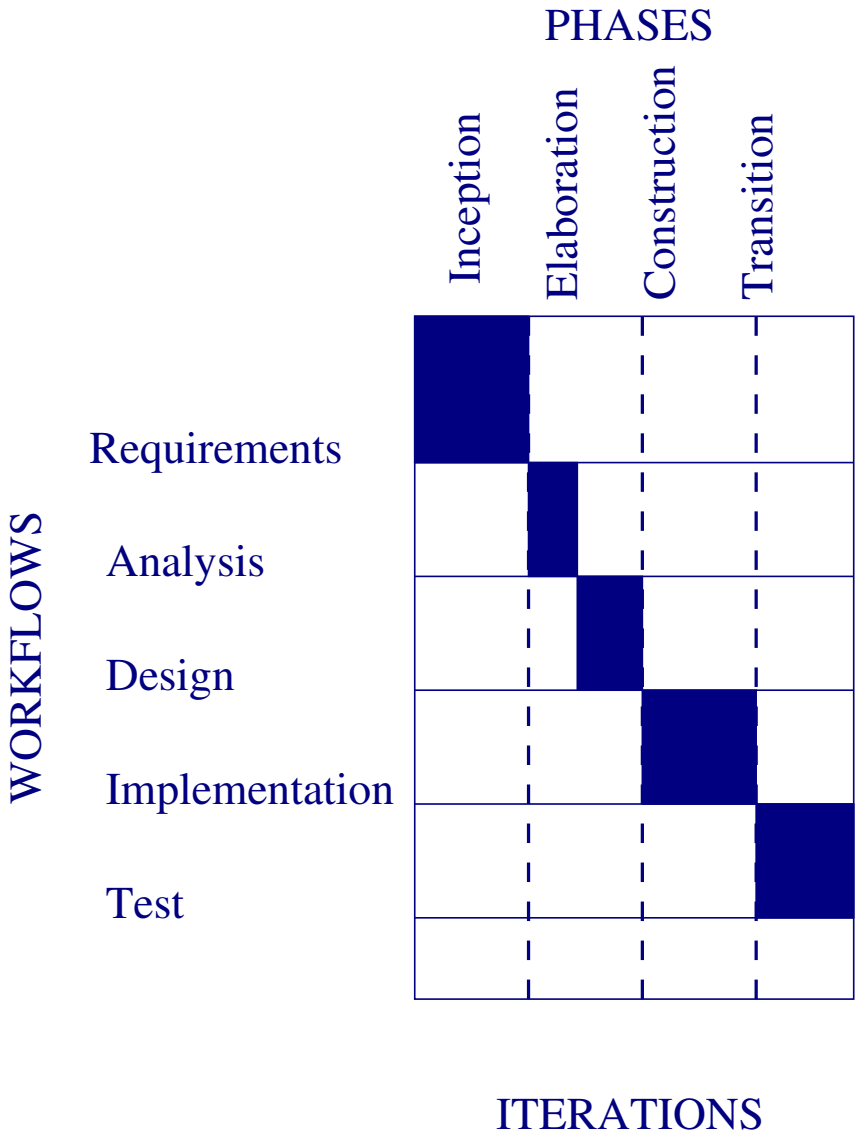
Waterfall Iterations Within Phases



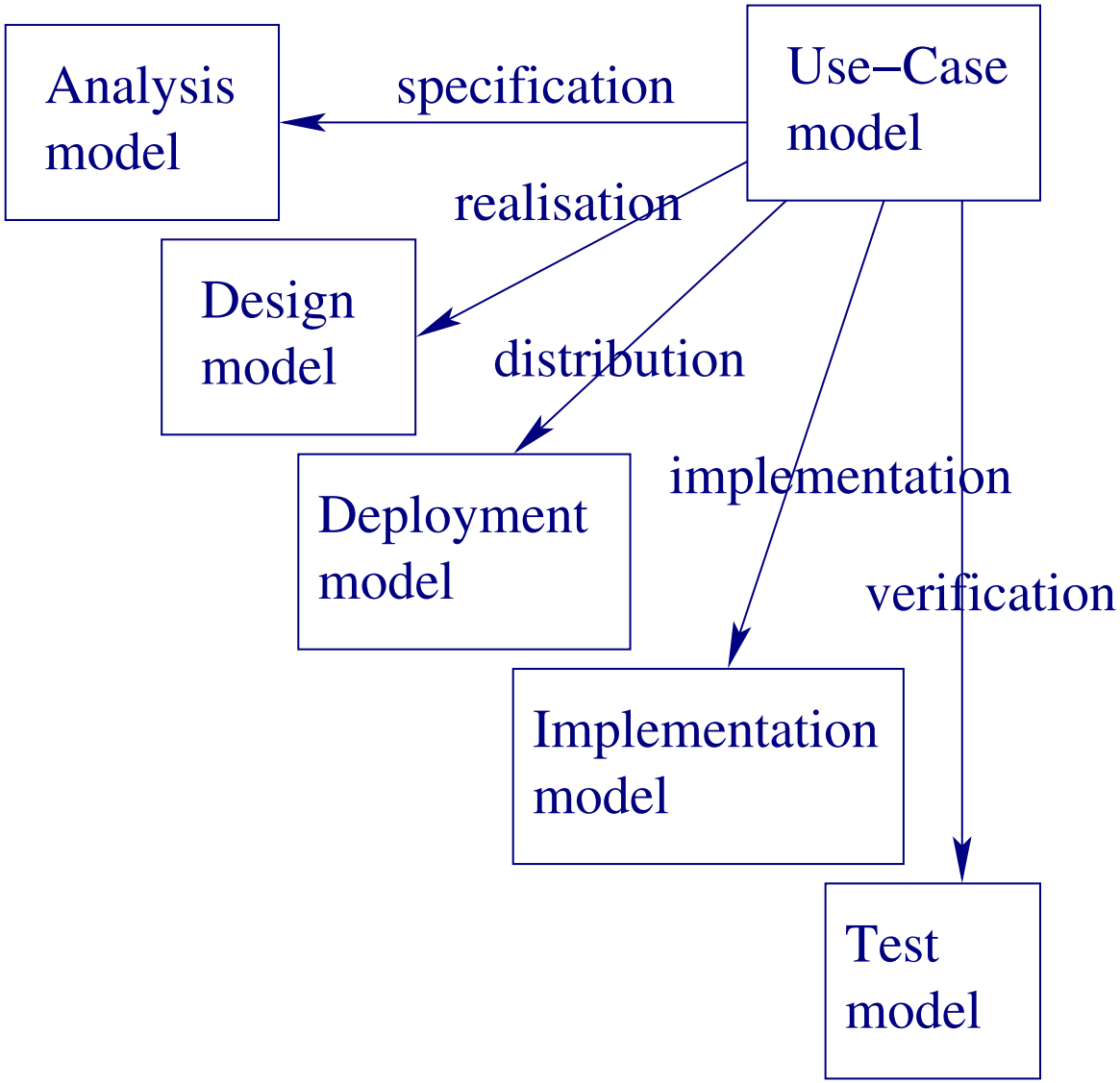
- Each phase can have multiple iterations
- Each iteration *can* include all workflows, but some are more heavily weighted in different phases
- Still hard to change requirements once implementation underway

UP vs. Waterfall Cycle





The Product: A Series of Models



Use Cases

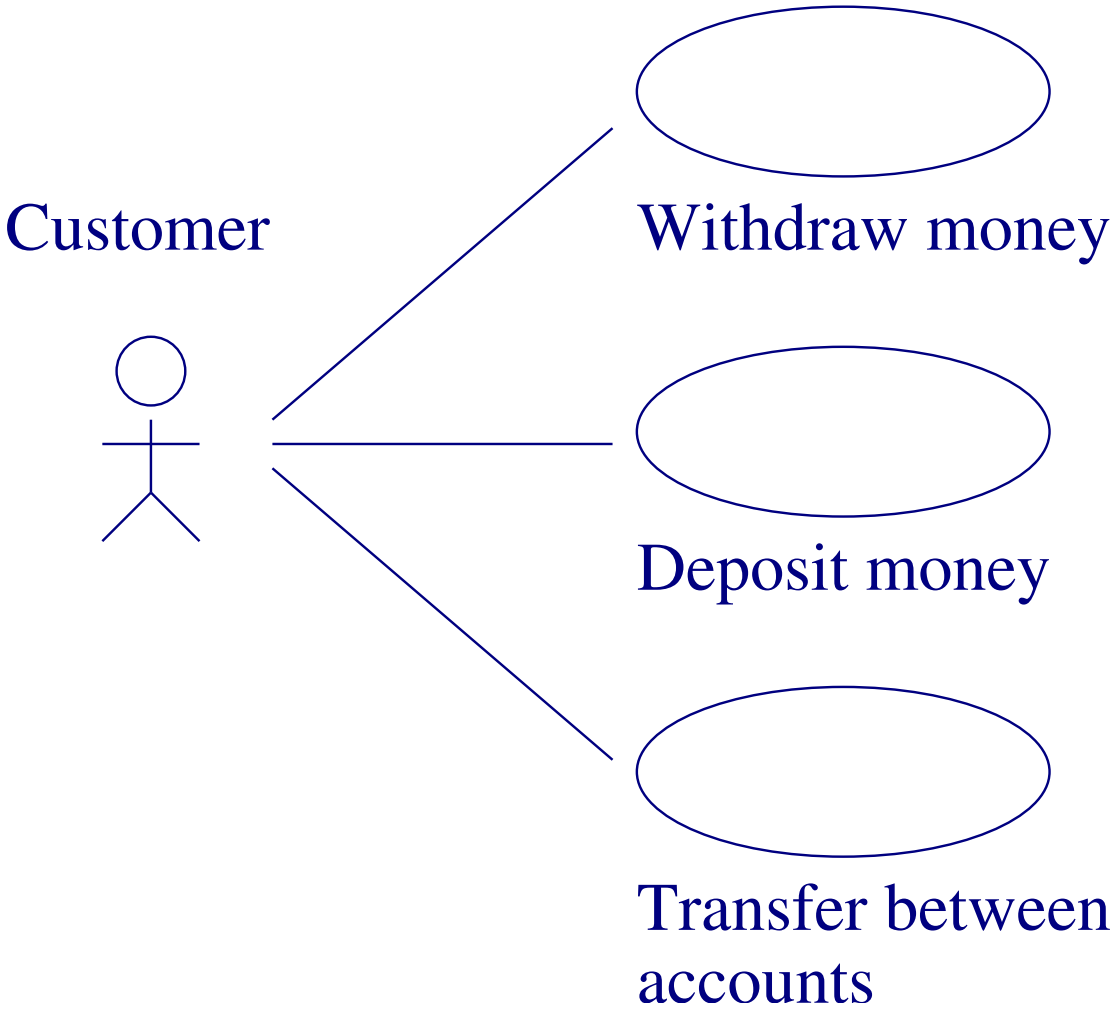
"A use case specifies a sequence of actions, including variants, that the system can perform and that yields an observable result of value to a particular actor."

These drive:

- Requirements capture
- Analysis and design of how system realizes use cases
- Acceptance/system testing
- Planning of development tasks
- Traceability of design decisions back to use cases

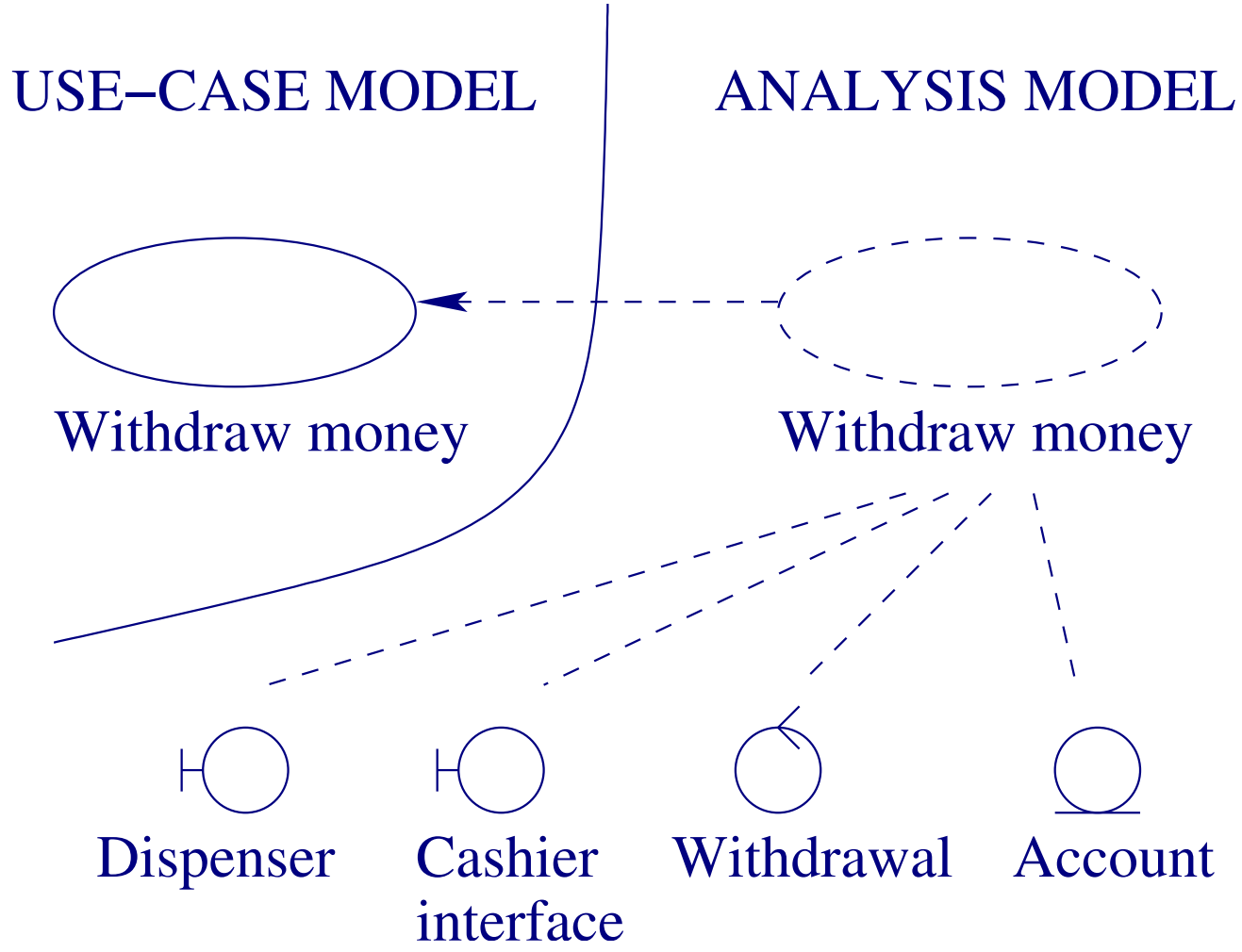
Use Case Example: 1

Initial use-case diagram:



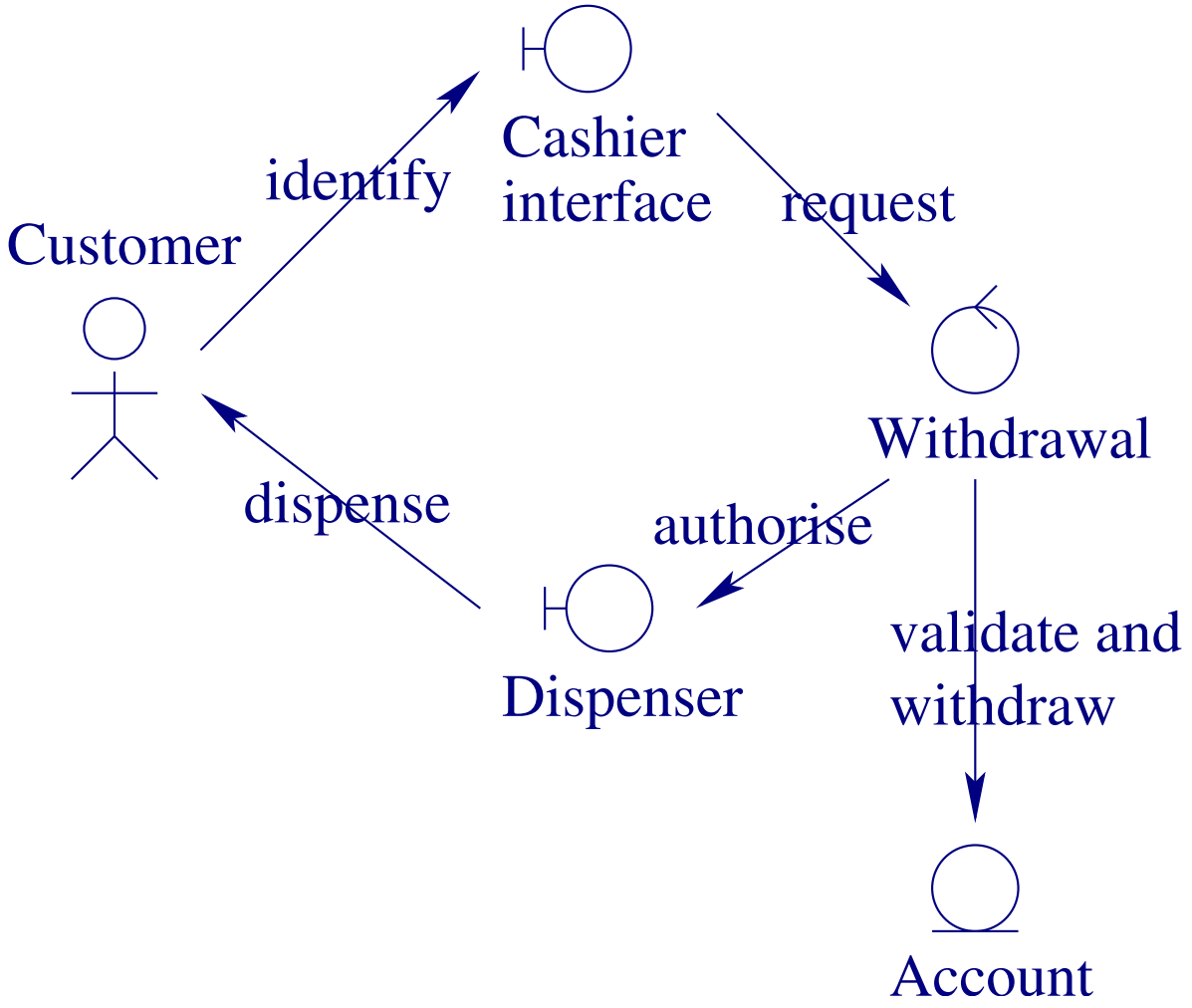
Use Case Example: 2

Analysis classes for withdrawing money:



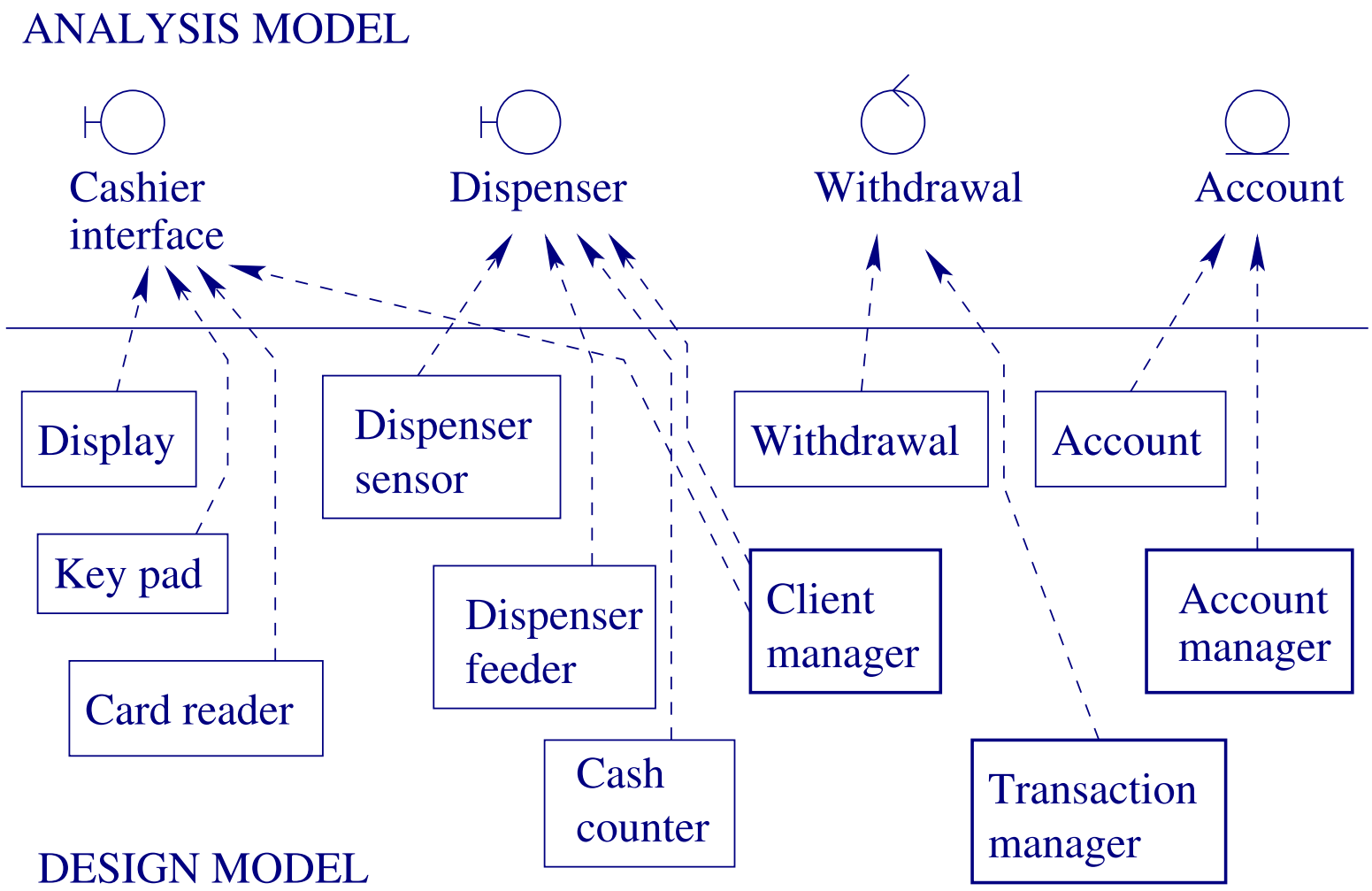
Use Case Example: 3

Collaboration diagram for withdrawing money:



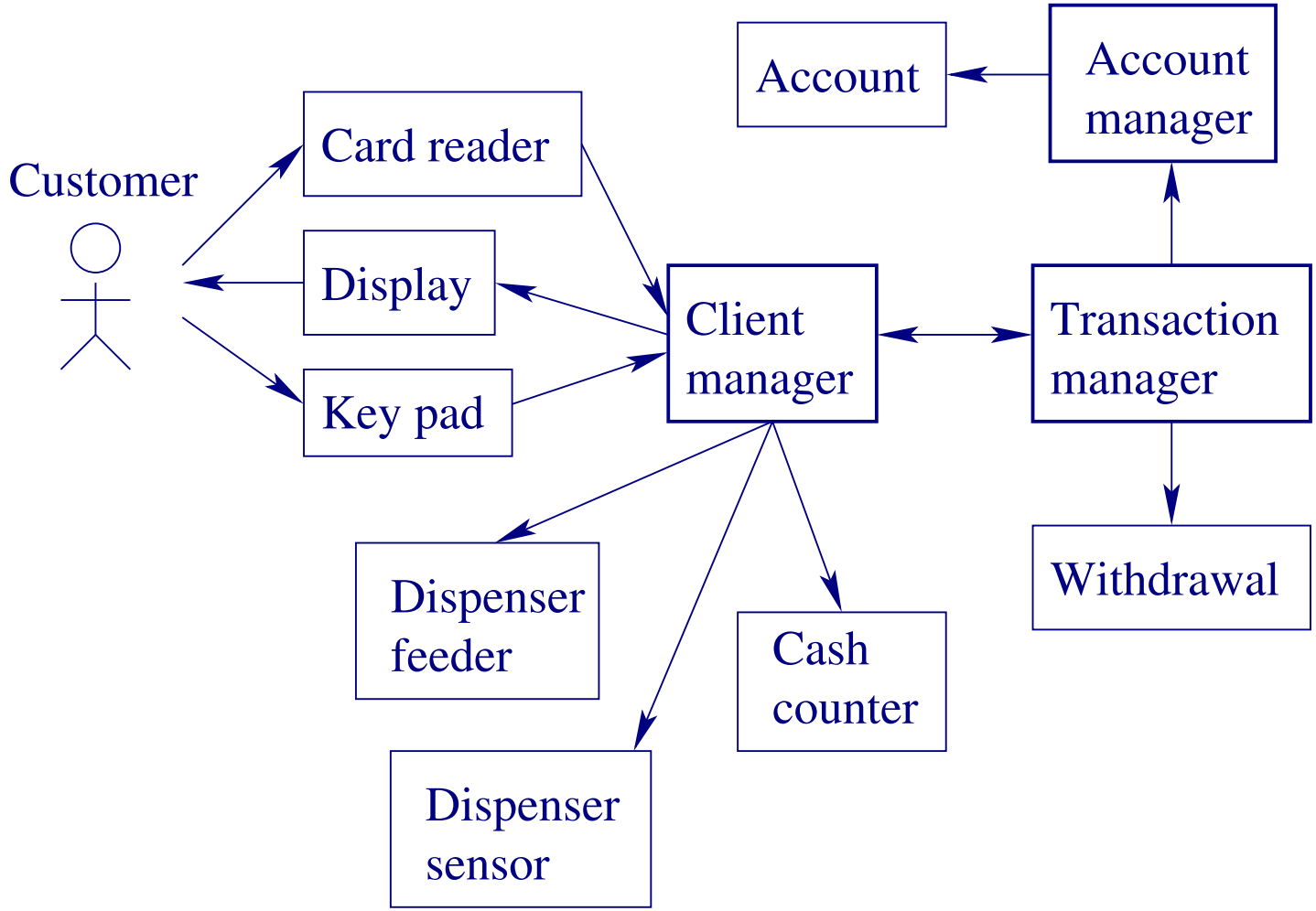
Use Case Example: 4

Design classes introduced for analysis classes:



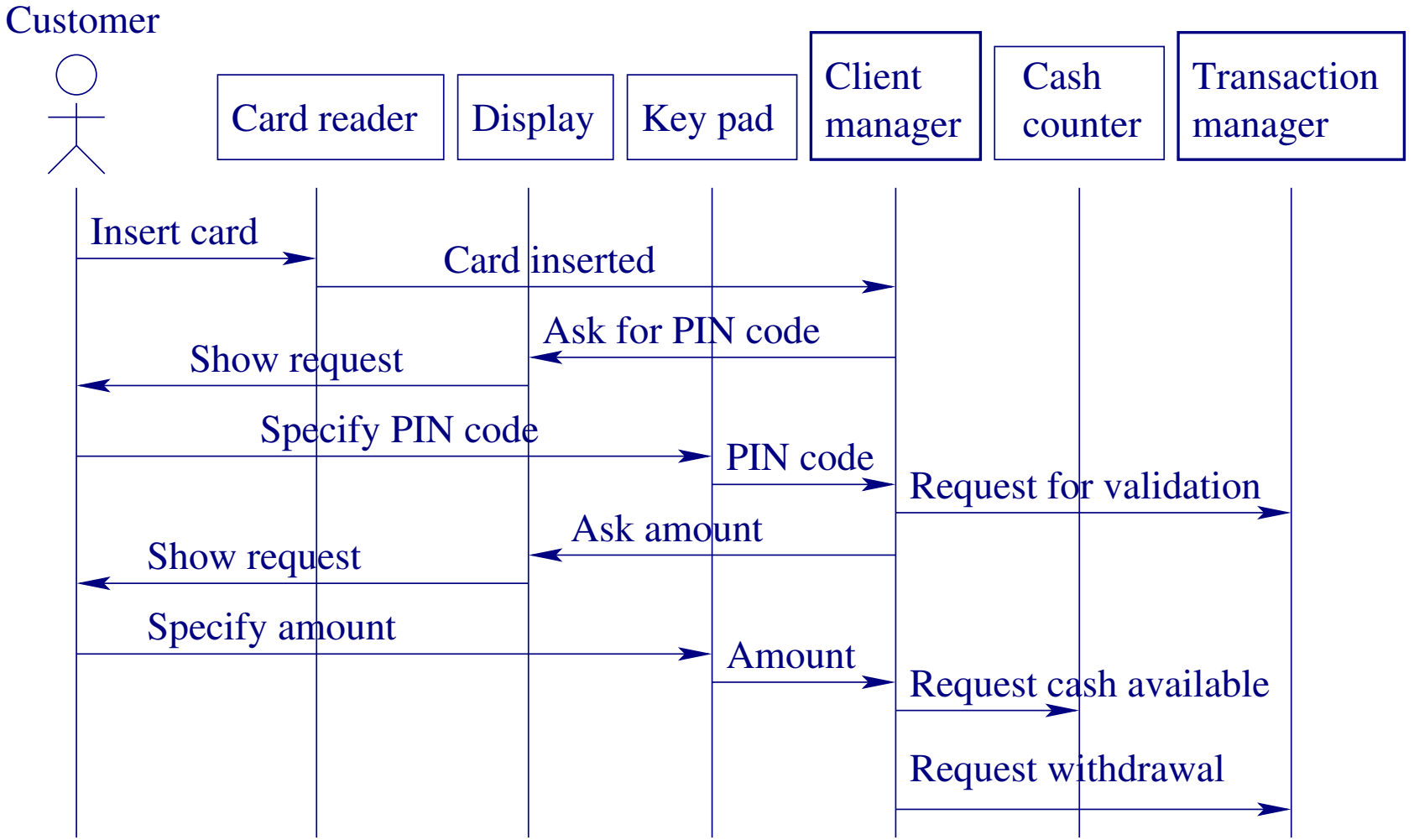
Use Case Example: 5

Class diagram which is part of the realization of the design model:



Use Case Example: 6

Sequence diagram for part of the realization:



Problems with UP

Heavy training, documentation, and tools requirements — learning and using UML, modeling, process, tools, techniques.

UML is not a native language for customers, and so they often cannot provide good feedback until system is implemented.

Requirements are very difficult to change at later stages, if needed to match changes in business world, address new competition, or fix mistakes in requirements capture.

Assumptions of UP

UP and other “heavyweight” methodologies concentrate on carefully controlled, up-front, documented thinking.

Based on assumption that cost of making changes rises exponentially through the development stages.

To minimize backtracking, establishes rigorous control over each stage.

At each stage a model acts as a proxy for the whole system, helping to bring out problems as early as possible (before they are set in code).

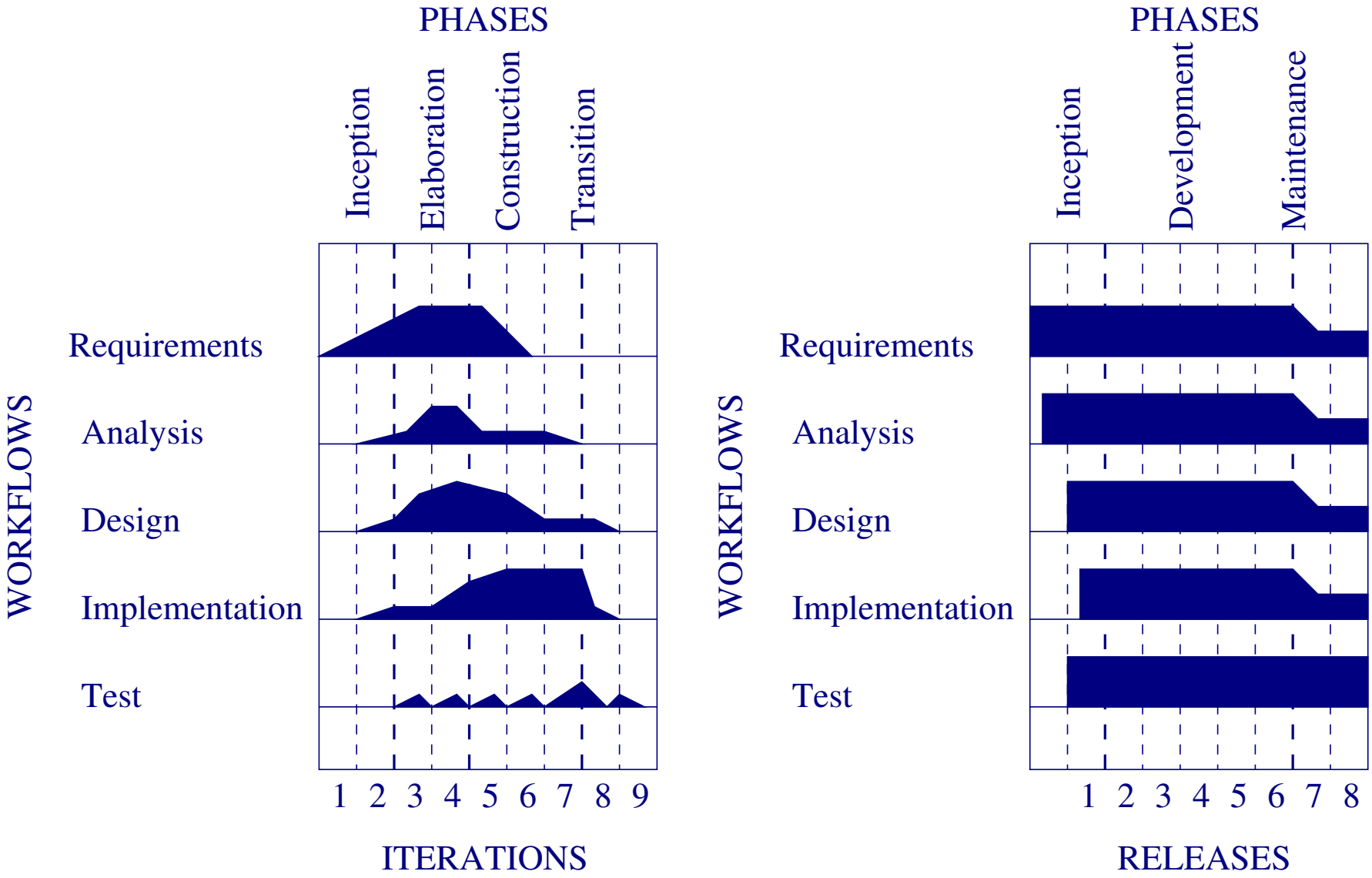
Extreme Programming (XP)

What if it were possible to make the cost of change constant across all stages, so that design and requirements can be changed even at late stages?

XP tries to prevent backtracking by keeping the system continuously flexible, eliminating the need for determining the final correct requirements and design before implementation.

XP is considered “lightweight”, and focuses on closely knit, fast moving (aka “agile”) design/coding teams and practices (Beck 1999).

UP Cycle vs. XP Development



XP is Controversial

An IBM Java poll on XP from

`www.xprogramming.com` said roughly this:

- “I’ve tried it and loved it” (51%)
- “I’ve tried it and hated it” (8%)
- “It’s a good idea but it could never work” (25%)
- “It’s a bad idea - it could never work” (16%)

Of course, the UP is widely resented as well...

How XP Imposes Control

Through a set of “practices” to which designers adhere (using whatever other compatible methods and tools they prefer). See: www.extremeprogramming.org/rules.html

Not strongly influenced by a particular design paradigm (unlike UP).

Does require a strongly held (“extreme”) view of how to approach design.

We consider some key practices in the following slides.

1. The Planning Process

An XP project starts with a “Planning Game”.

The “customer” defines the business value of desired “user stories”.

The programmers provide cost estimates for implementing the user stories in appropriate combinations.

No one is allowed to speculate about producing a total system which costs less than the sum of its parts.

User Stories vs. Use Cases

A user story meets a similar need as a use case, but is textual, not graphical, and is something that any customer can do without training in UML.

A user story deliberately does not include all the possible exceptions, variant pathways, etc. that go into use cases.

Short example: “A bank customer goes up to an ATM and withdraws money from his or her account.”

2. On-site customer

Someone who is knowledgeable about the business value of the system sits with the design team.

This means there is always someone on hand to clarify the business purpose, help write realistic tests, and make small scale priority decisions.

The customer acts as a continuously available source of corrections and additions to the requirements.

3. Small Releases

Put a simple system into production early, implementing a few important user stories.

Re-release it as frequently as possible while adding significant business value (a set of important user stories) in each release. E.g., aim for monthly rather than annual release cycles.

The aim is to get feedback as soon as possible.

4. Continuous Testing

Write the tests before writing the software.

Customers provide acceptance tests.

Continuously validate all code against the tests.

Tests act as system specification.

5. Simple Design

Do the simplest thing that could possibly work.

Don’t design for tomorrow — you might not need it.

Extra complexity added “just in case” will fossilize your design (e.g. your class hierarchies) and get into the way of the changes you will need to make tomorrow.

6. Refactoring

When tomorrow arrives, there will be a few cases where you actually have to change the early simple design to a more complicated one.

Change cannot occur only through small, scattered changes, because over time such changes will gradually turn the design into spaghetti.

To keep the design modifiable at all stages, XP relies on continuous refactoring: improving the design without adding functionality.

Refactoring Approach

Whenever the current design makes it unwieldy to implement the current user story:

1. Step back and re-design the existing code so that it will make the change easy and clean.
2. Make sure that the code meets the same tests as before, i.e., provides the same functionality.
3. Integrate the changes with the team.
4. Make the change, pass the tests, and integrate again.

Refactoring Guideline

“Three strikes and you refactor” principle - e.g. consider removing code duplication if:

- The 1st time you need the code you write it
- The 2nd time, you reluctantly duplicate it
- The 3rd time, you refactor and share the resulting code

Refactoring requires a system for integrating changes from different teams.

7. Collective Ownership

Anyone is allowed to change anyone else's code modules, without permission, if he or she believes that this would improve the overall system.

To avoid chaos, collective ownership requires a good configuration management tool, but those are widely available.

8. Coding Standard

Since XP requires collective ownership (anyone can adapt anyone else's code) the conventions for writing code must be uniform across the project.

This requires a single coding standard to which everyone adheres.

9. Continuous Integration

Integration and full-test-suite validation happens no more than a day after code is written.

This means that individual teams don’t accumulate a library of possibly relevant but obscure code.

Moreover, it enables everyone to freely modify code at any time, because they know that they have access to the latest design.

10. Pair Programming

All code is written by **a pair** of people at one machine.

- One partner is doing the coding
- The other is considering strategy (Is the approach going to work? What other test cases might we need? Could we simplify the problem so we don't have to do this? Etc.)

This is unpalatable to some but appears vital to the XP method, because it helps make collective code ownership work.

11. 40-Hour week

XP is intense so it is necessary to prevent “burnout”.

Designers are discouraged from working more than 40 hours per week.

If it is essential to work harder in one week then the following week should drop back to normal (or less).

Problems with XP

Published interfaces (e.g. APIs): some code is not practical to refactor, because not all uses can be known, so that code must anticipate all reasonable tomorrows.

Many programmers resist pair programming or other XP guidelines; teams are often spread geographically, and even at one site sharing a computer is often awkward.

The customer isn't always available or willing, and may not be able to agree to an open-ended process.

Over time XP has become more heavy weight, trying to incorporate new realizations, just as UP did.

Summary

- Methodologies: principled ways to manage large projects
- Waterfall model works in other disciplines, where most of the work is on the physical implementation, but in SE all work is conceptual
- Unified Process constructs gradually more elaborate models to uncover risks and solidify requirements and design as early as possible
- Extreme Programming relies on continuous customer involvement, testing, and refactoring to deliver code early and continuously, minimizing risk of complete failure.

References

Beck, K. (1999). *Extreme Programming Explained*. Reading, MA: Addison-Wesley.

Jacobson, I., Booch, G., & Rumbaugh, J. (1998). *The Unified Software Development Process*. Reading, MA: Addison-Wesley.

Royce, W. W. (1970). Managing the development of large software systems. In *Proceedings of IEEE WESCON*, pp. 1–9.