Unified Modeling Language, UML 2.x cont.

Abdulkareem Alali

ACK Jonathan Maletic, Grady Booch, IBM Rational Software, Martin Fowler, …

What Is Generalization?

One class shares the structure and/or behavior of one or more classes

A subclass inherits its parent's attributes, operations, and relationships

A subclass add additional attributes, operations, relationships, redefine inherited operations

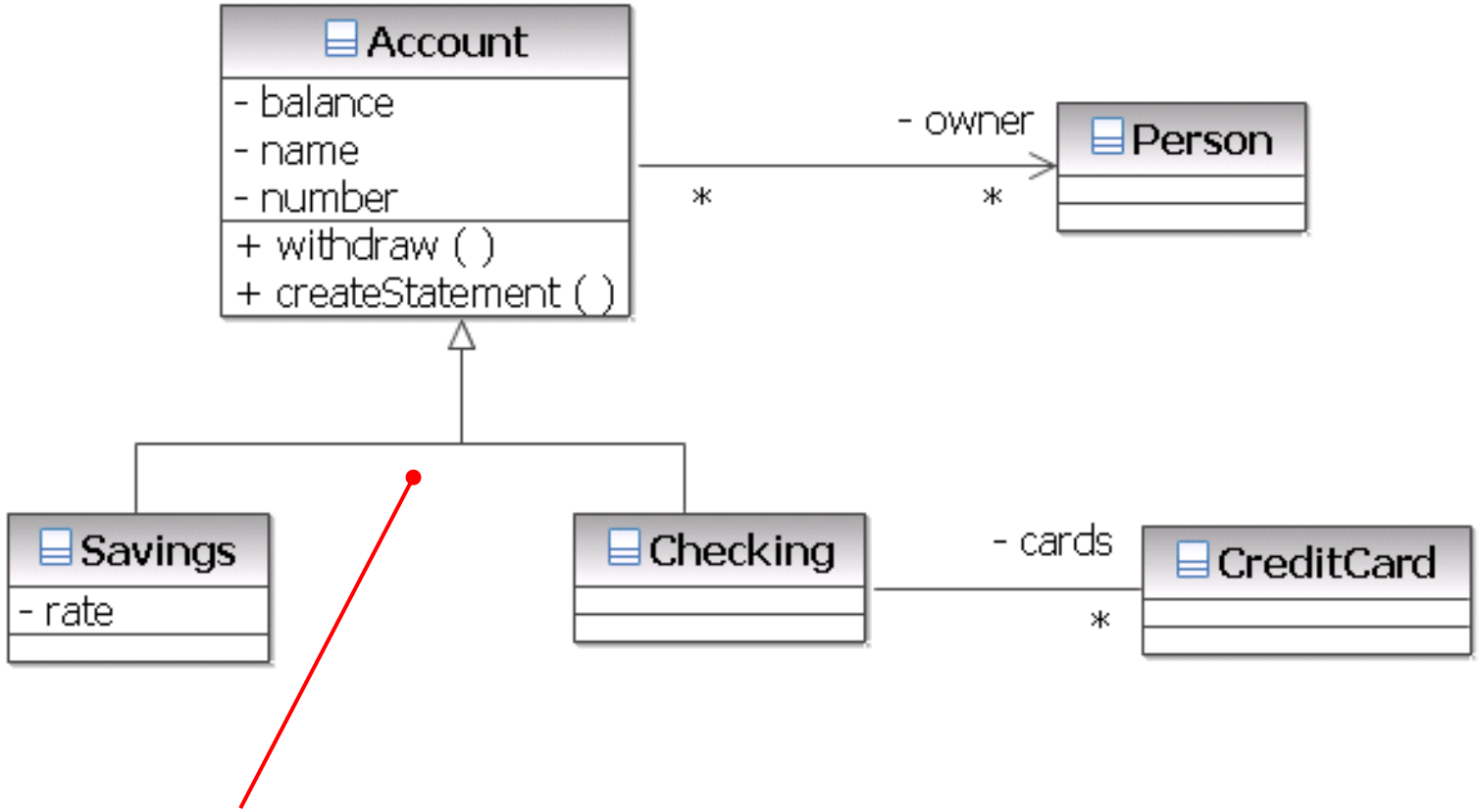
"**is a**" relationship

Single or multiple inheritance

Example

Superclass, Parent or Ancestor

Subclasses, Children or Descendants

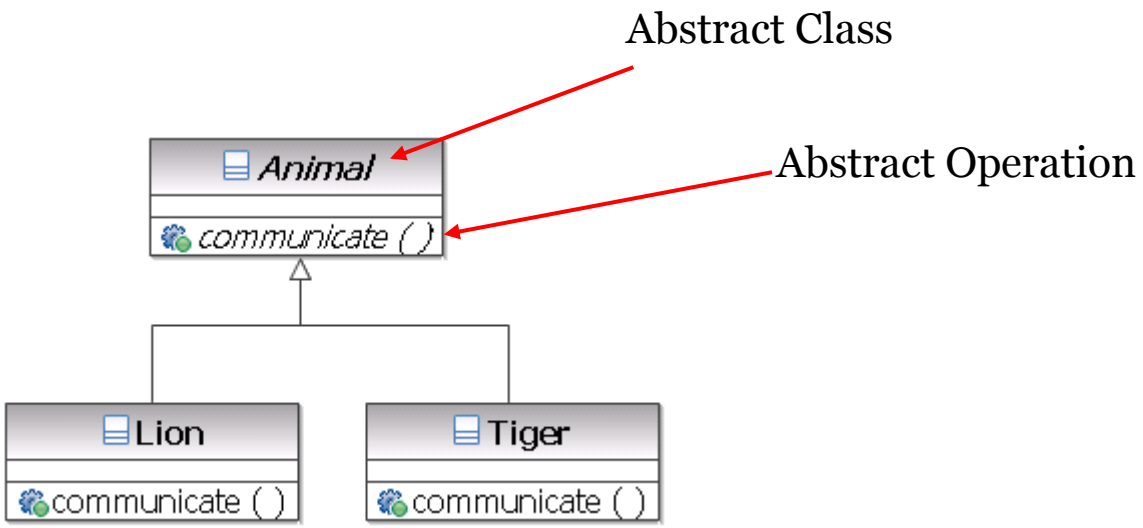


Generalization Relationship

What is an Abstract Class?

Abstract classes cannot be instantiated, Name in *italics*

Abstract operations: the subclasses must provide the implementation



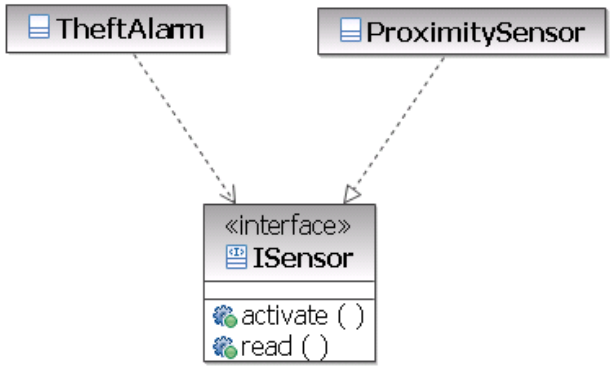
All objects are either lions or tigers

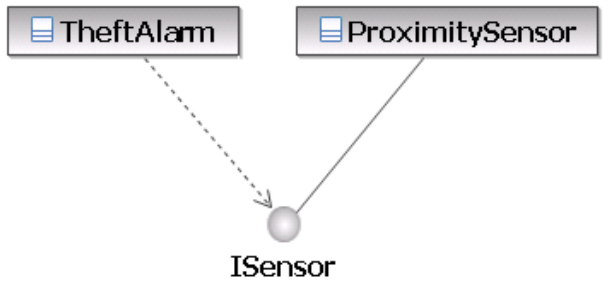
What Is An Interface? .

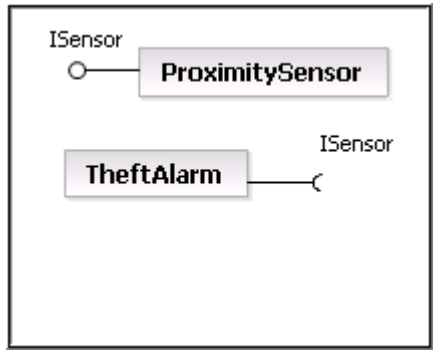
A “contract” between providers and consumers of services

Equivalent to an **abstract class** where all operations are abstract

- Provided interfaces: interfaces the element exposes to its environment
- Required interfaces: interfaces the element requires from other elements
- Note: UML interfaces can have attributes







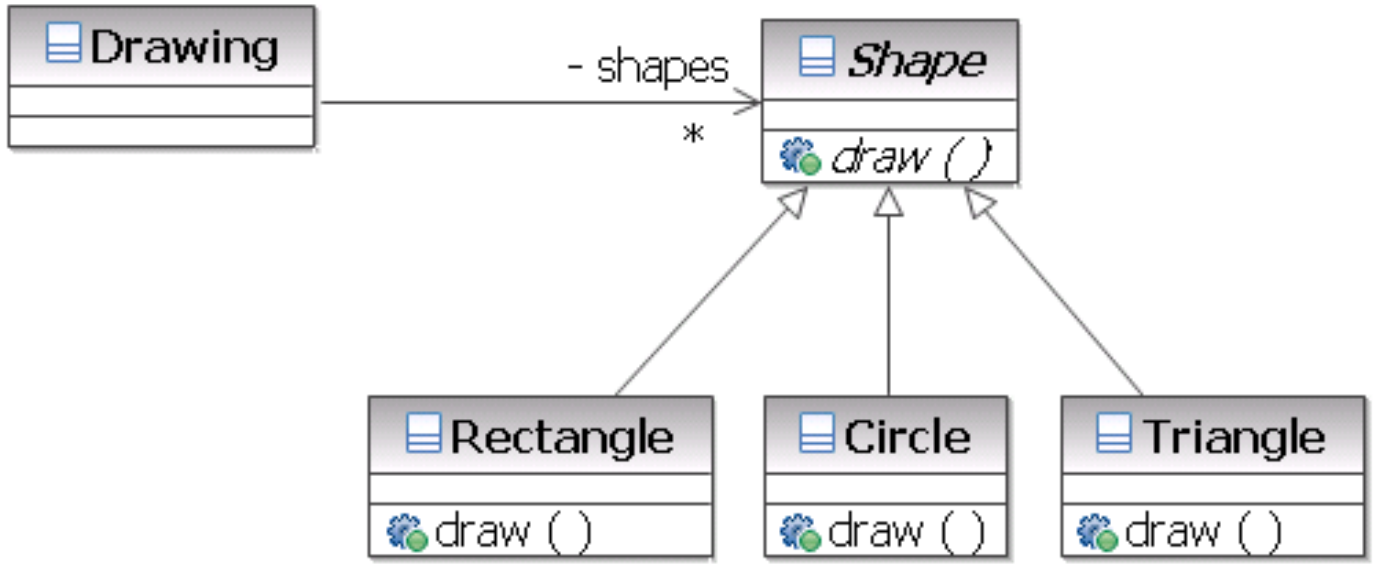
Ball and Socket notation

What Is Polymorphism?

The ability to hide different implementations behind a single class or interface

```
Polymorphism

Shape s = …;
s.draw();
```

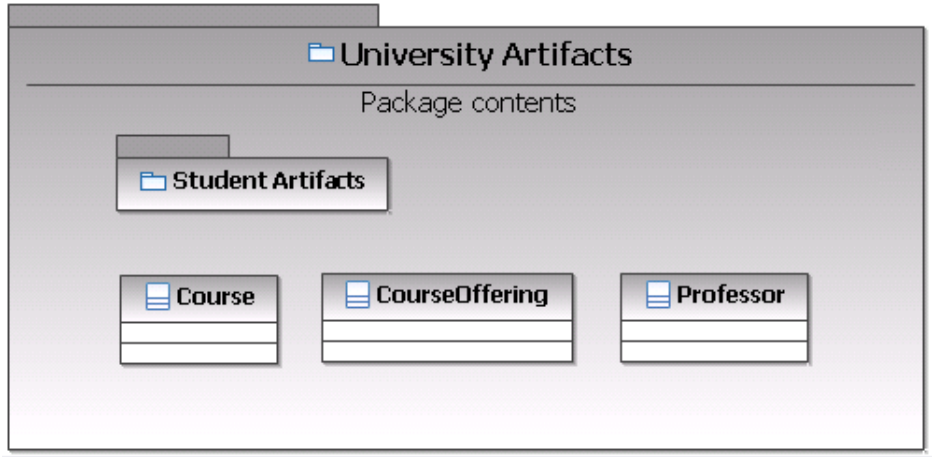


What Is a Package?

A general-purpose mechanism for organizing elements into groups

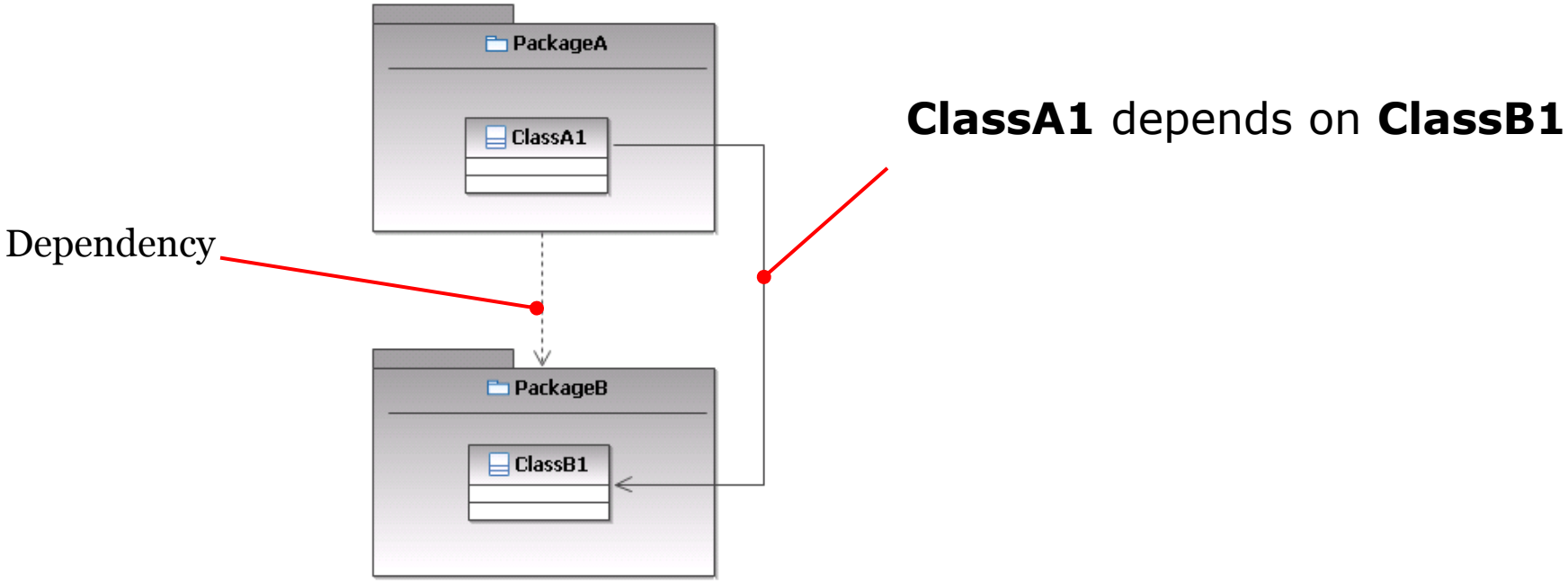
A model element that can contain other model elements

- Grouping of logically related elements



Package Relationships

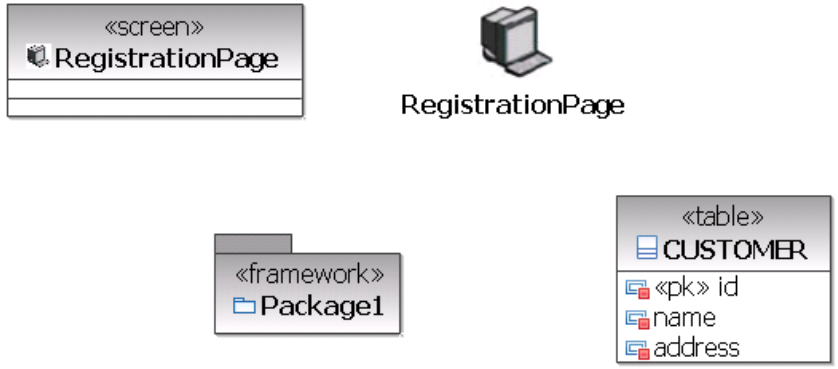
A package **A** depends on a package **B** if at least one element from **A** depends on at least one element from **B**



What Is A Stereotype?

A stereotype is a mechanism used to extend the vocabulary of UML

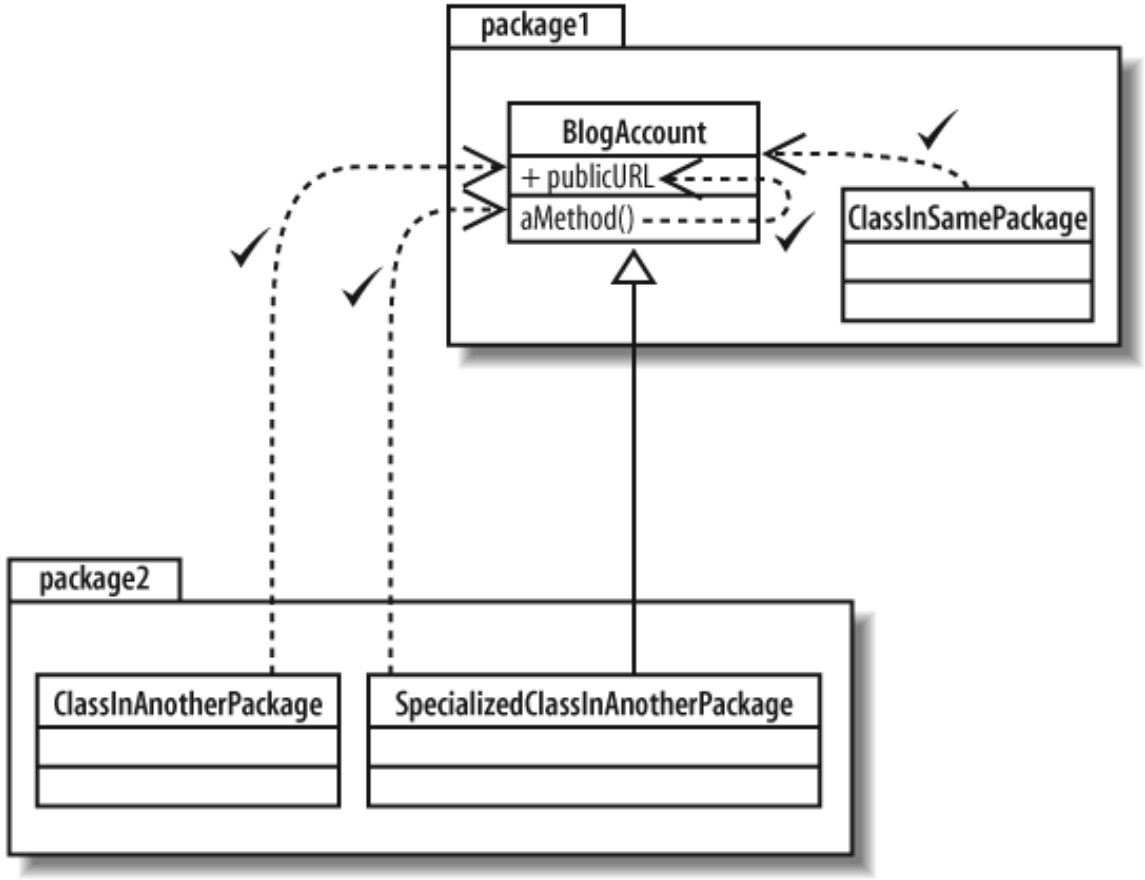
Represented textually (`<<mystereo>>`) and/or graphically



Any UML element may be stereotyped

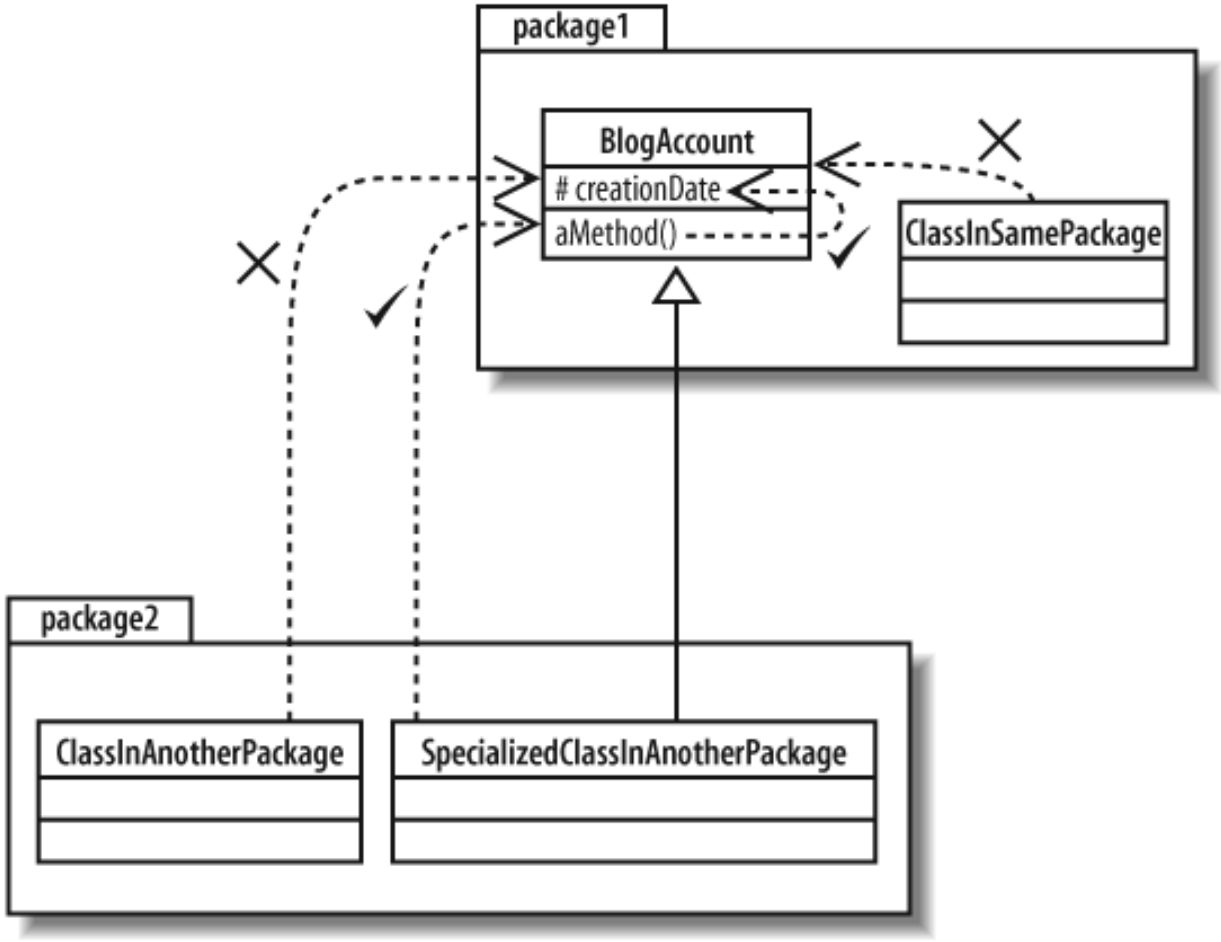
Visibility: public . .

(+): an attribute or operation accessible directly by any other class



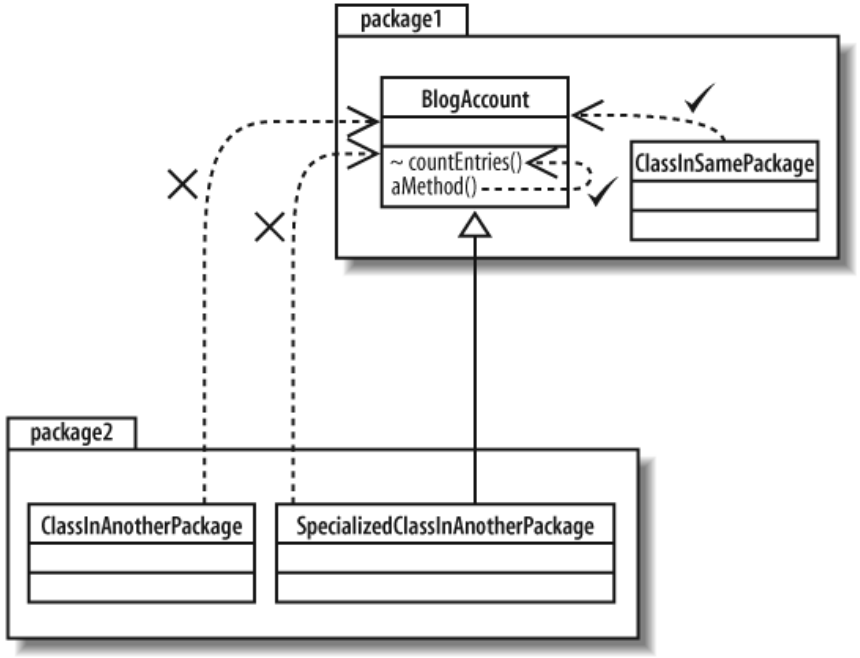
Visibility: protected . .

(#) : classes methods and attributes can be accessed by methods of it's direct or indirect subclasses



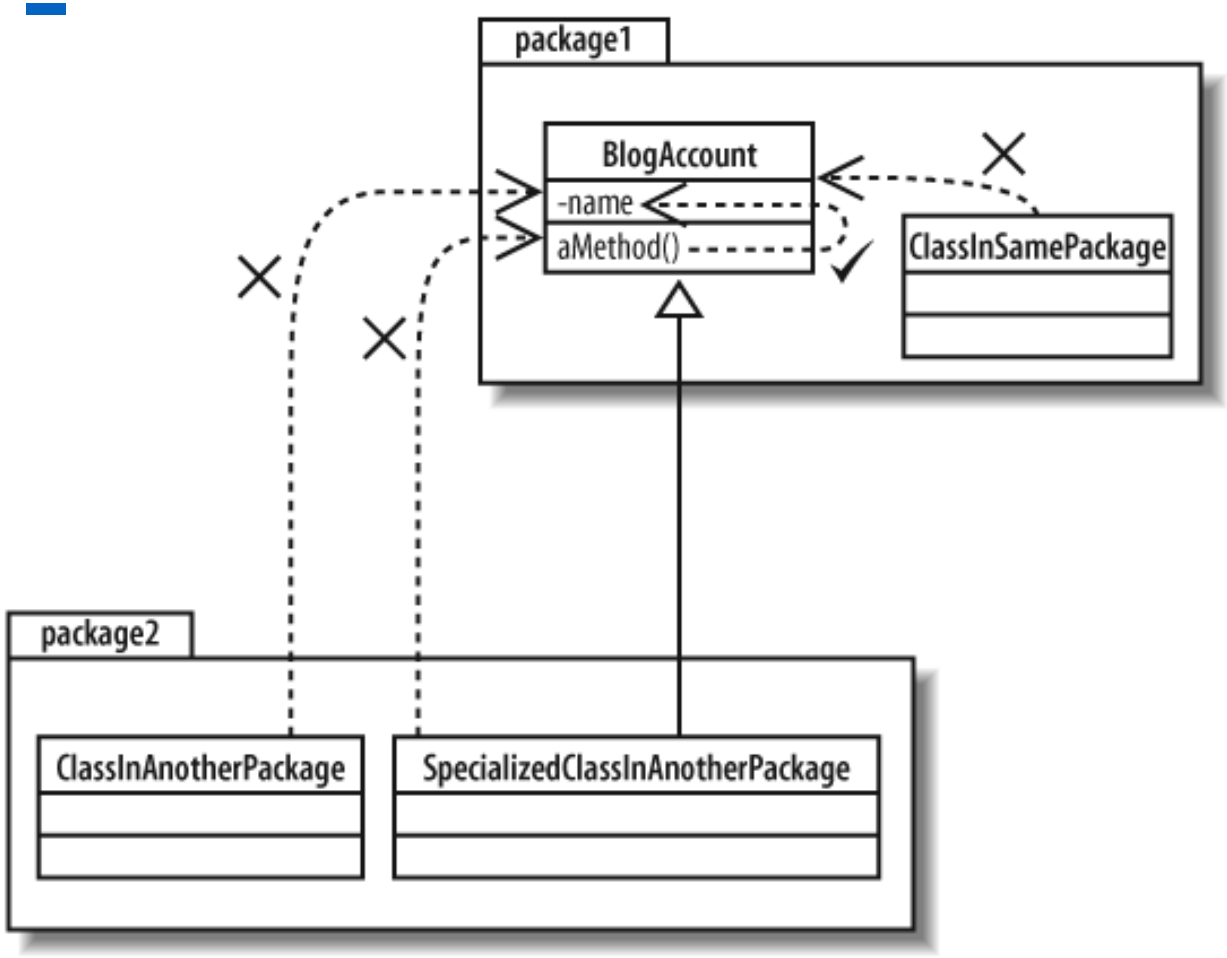
Visibility: package . .

(~) : An attribute or operation declared with package visibility in a class, any class in the same package can directly access it



Visibility: private . .

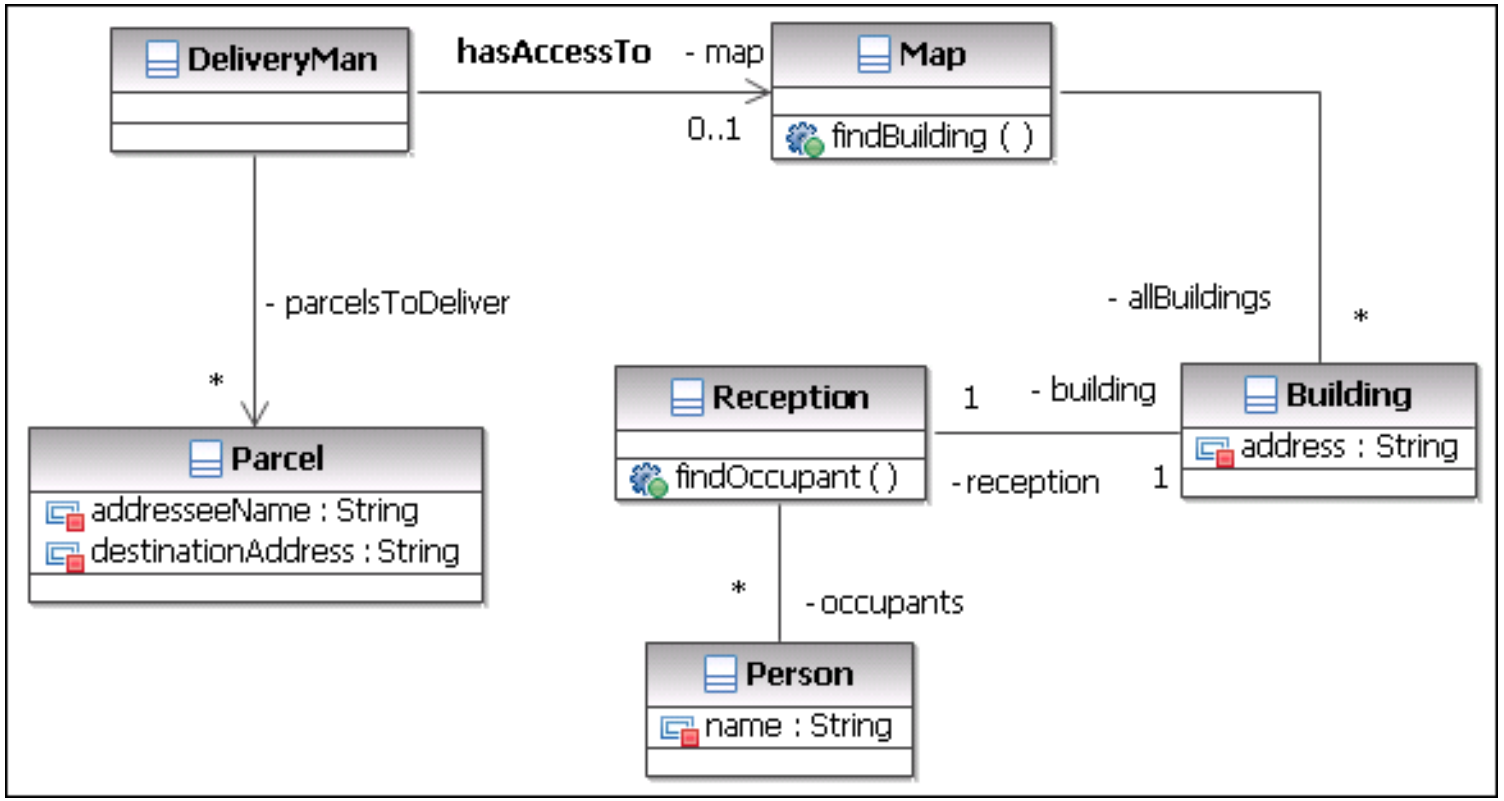
(-) : Only the class that contains the private element can see or work with the data stored in a private attribute or make a call to a private operation



Object Diagram

A snapshot of the system at a given time, highlighting existing and non-existing links

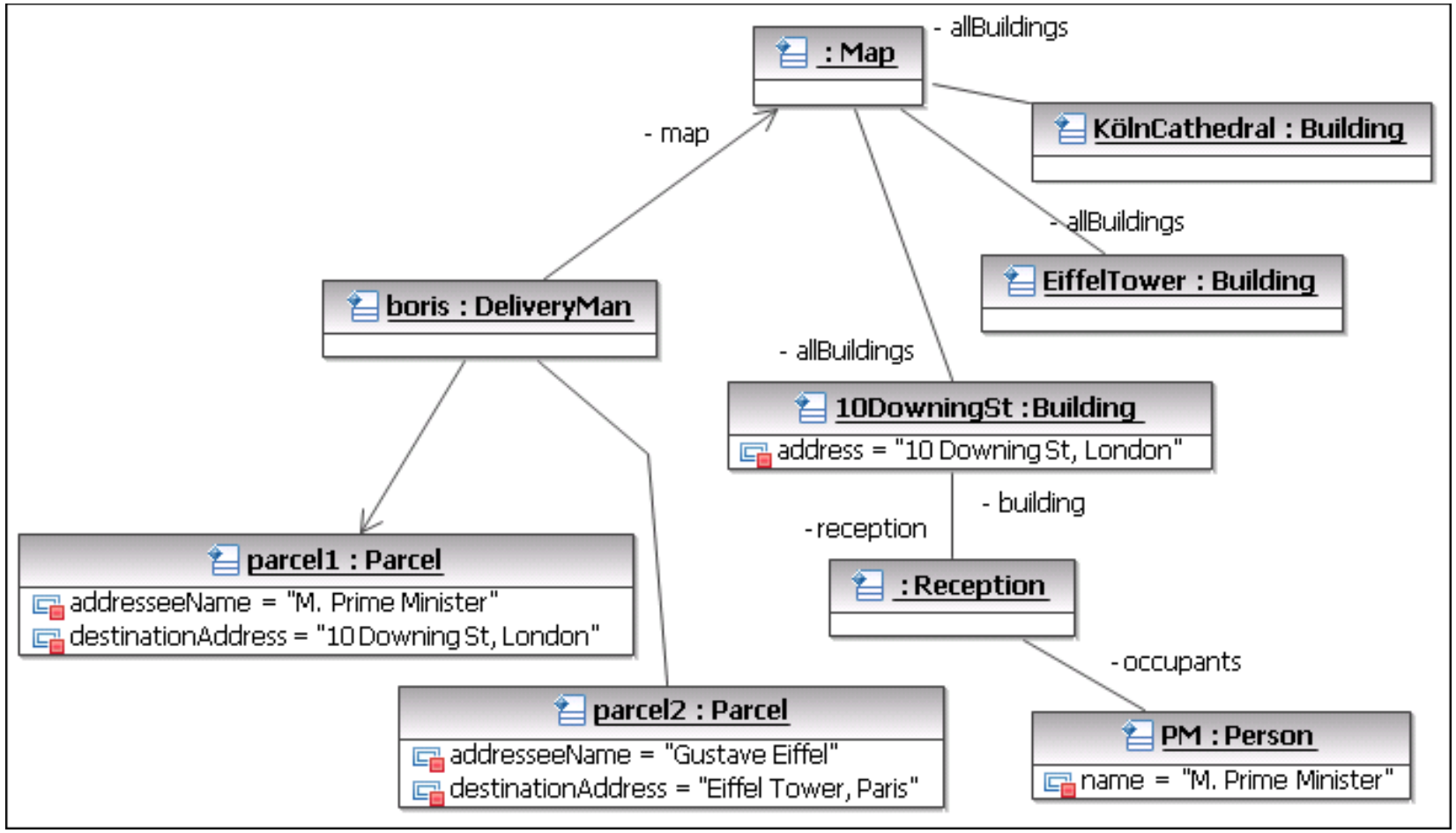
- Here ***boris*** needs to establish a link with ***PM*** to deliver *parcel1* …



Class Diagram

Object Diagram

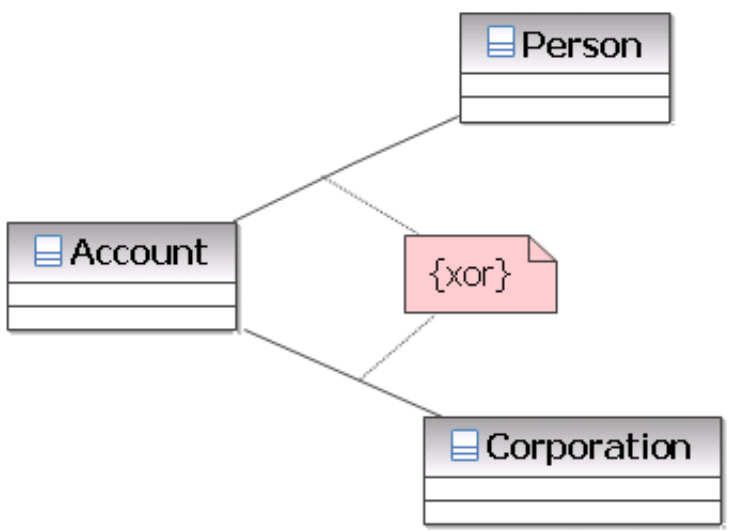
Object diagrams conceptualize abstract class diagrams by providing “real-world” examples of objects and object connections

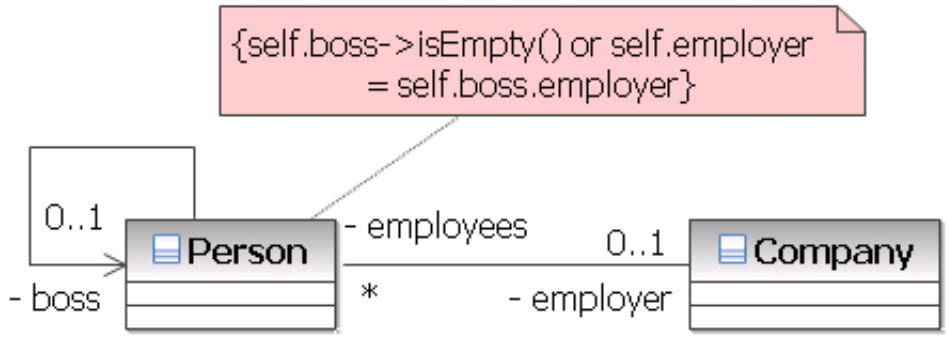


What Is A Constraint?

A semantics condition or restriction (a Boolean expression) expressed in natural language text or in a machine-readable language, some are predefined in UML ("xor"), and others may be user-defined.

OCL (Object Constraint Language) is a predefined language for writing constraints





UML Part IV

Modeling Behavior

- Interaction Diagrams
- State Chart Diagrams
- Activity Diagrams

Refining the Object Model

Typically, very simplistic object models can be directly derived from use cases

A better understanding of the behavior of each use case is necessary (i.e., analysis)

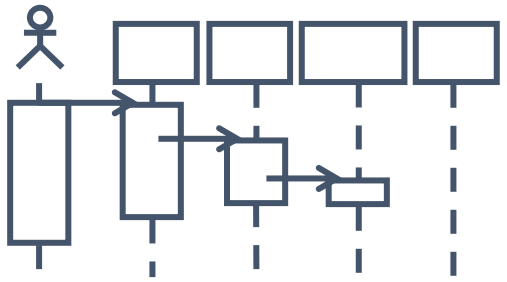
Use interaction diagrams to specify and detail the behavior of use cases

This helps to identify and refine key abstractions and relationships

Operations, attributes, and messages are also identified during this process

What Is a Sequence Diagram?

A sequence diagram is an interaction diagram that emphasizes the time ordering of messages and objects participating in the interaction



Sequence Diagrams

Sequence Diagrams

X-axis is objects

- Object that initiates interaction is left most
- Object to the right are increasingly more subordinate

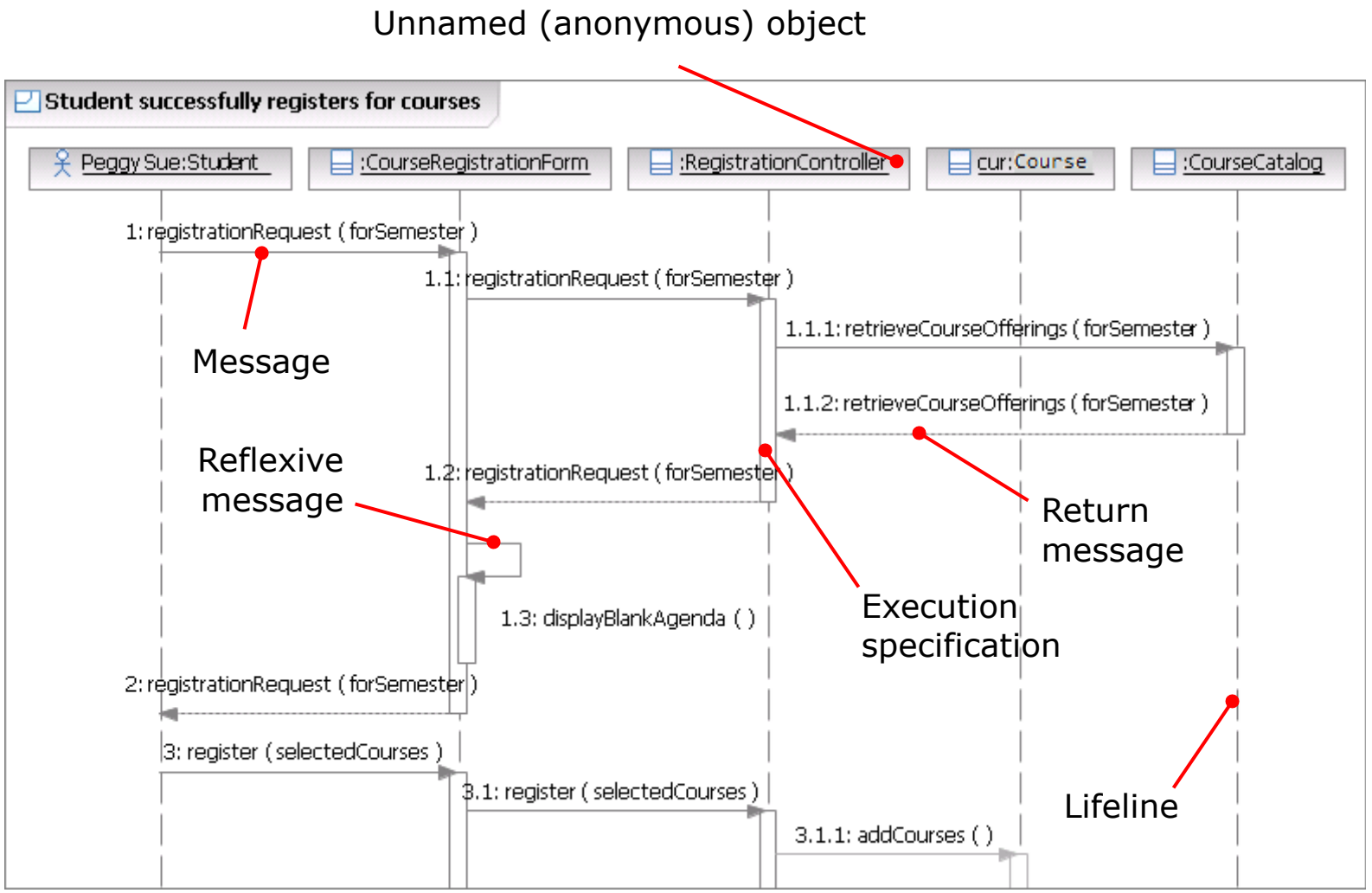
Y-axis is time

- Messages sent and received are ordered by time

Object lifelines represent the existence over a period

Activation (double line) is the execution of the procedure

Sequence Diagrams: Example



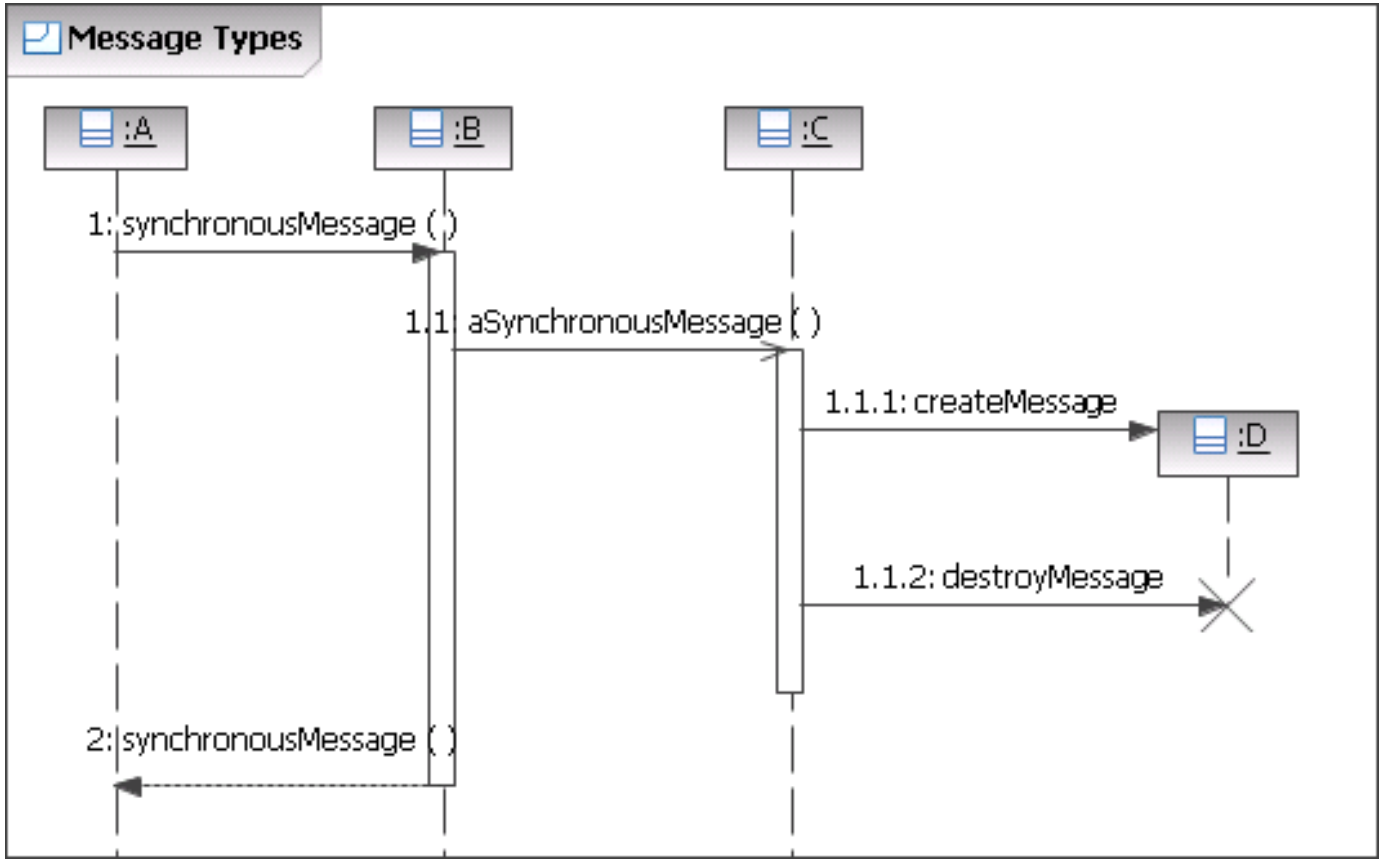
Message Types

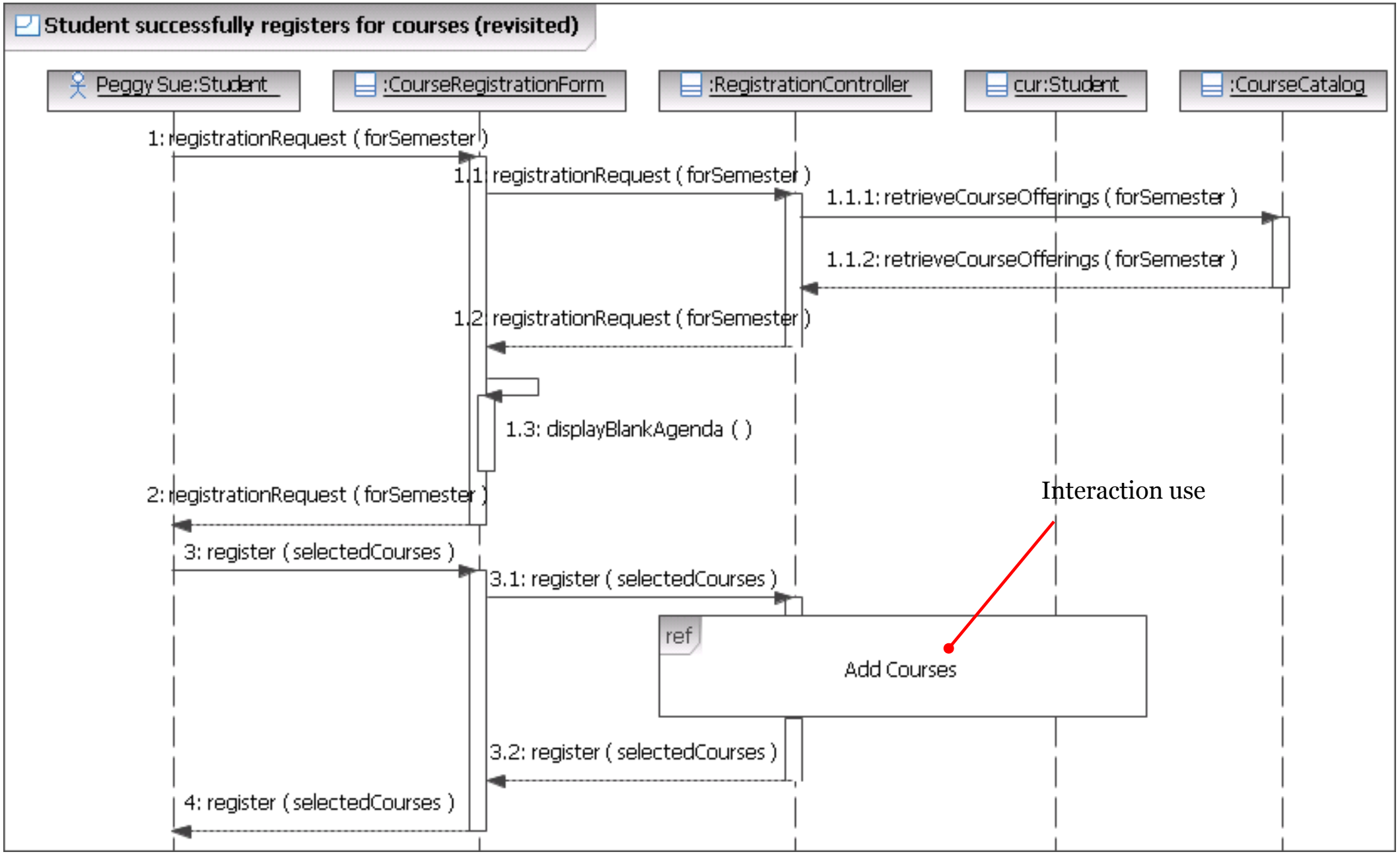
Synchronous message

Asynchronous message

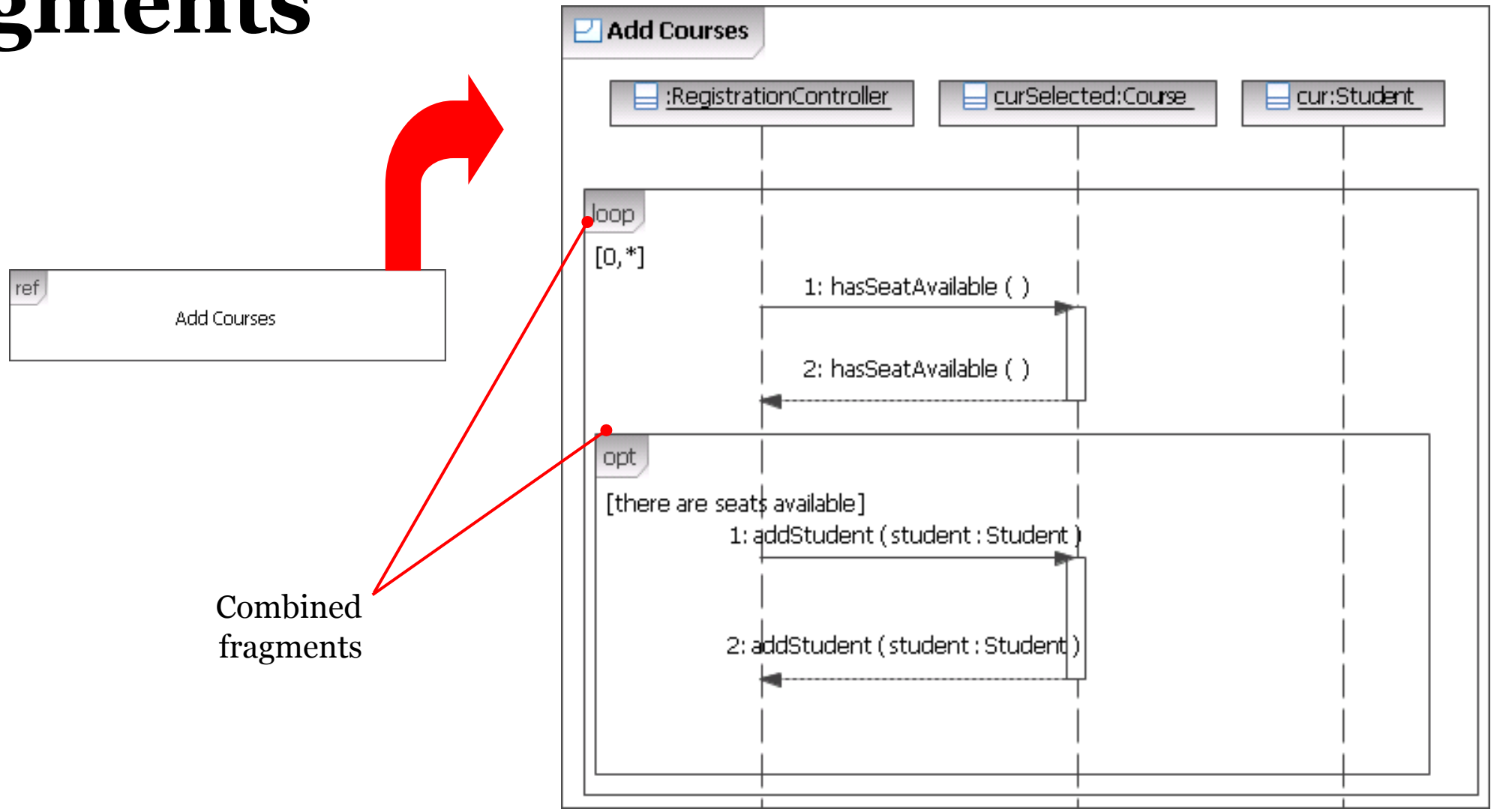
Create Message

Delete Message

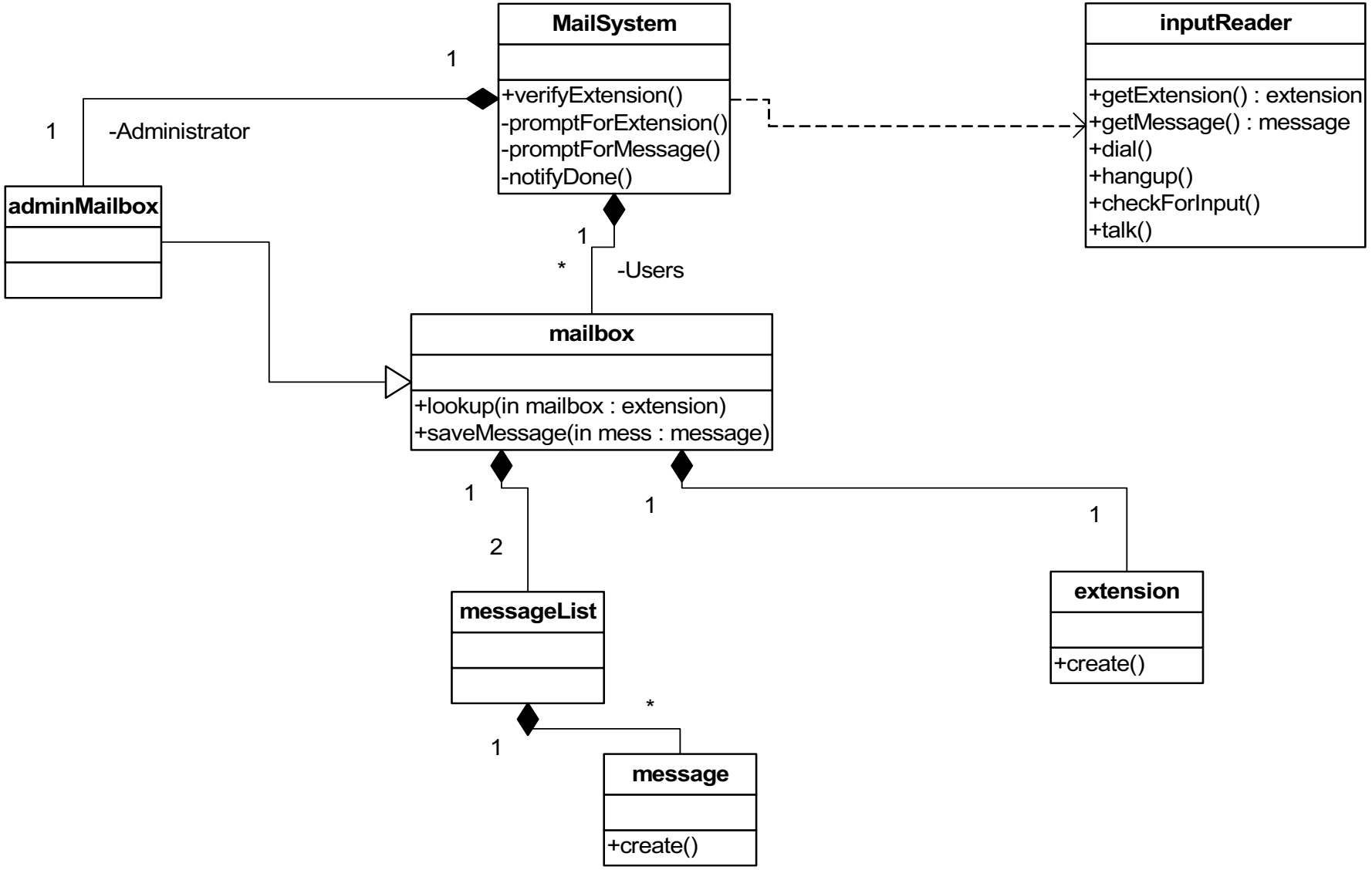



Student successfully registers for courses (revisited)
Peggy Sue:Student
:CourseRegistrationForm
:RegistrationController
cur:Student
:CourseCatalog
1: registrationRequest (forSemester)
1.1 registrationRequest (forSemester)
1.1.1: retrieveCourseOfferings (forSemester)
1.1.2: retrieveCourseOfferings (forSemester)
1.2 registrationRequest (forSemester)
1.3: displayBlankAgenda ( )
2: registrationRequest (forSemester)
3: register (selectedCourses)
3.1: register (selectedCourses)
ref
Add Courses
Interaction use
3.2: register (selectedCourses)
4: register (selectedCourses)

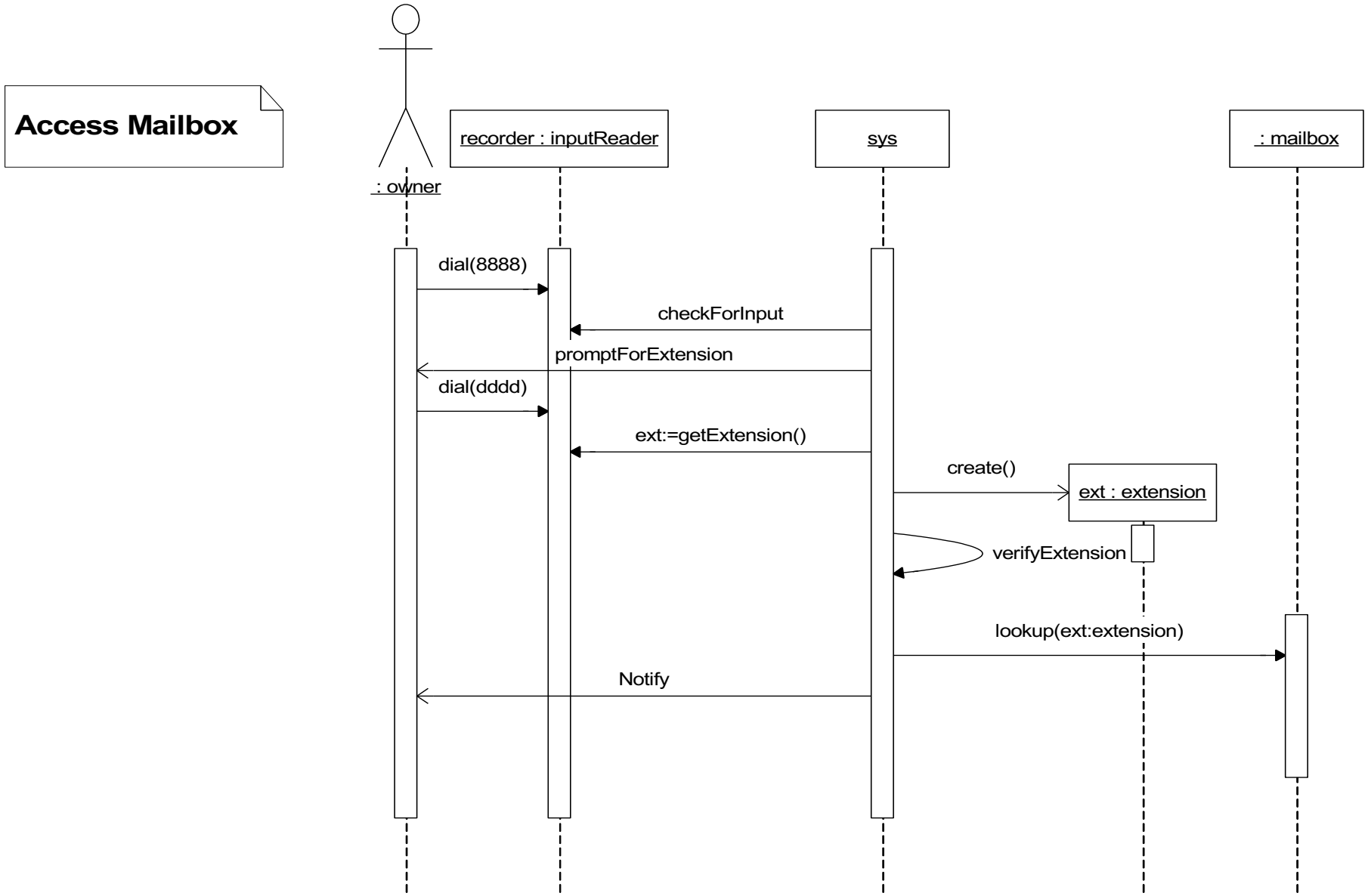
Interaction Use and Combined Fragments



UML Class Diagram: Mailbox

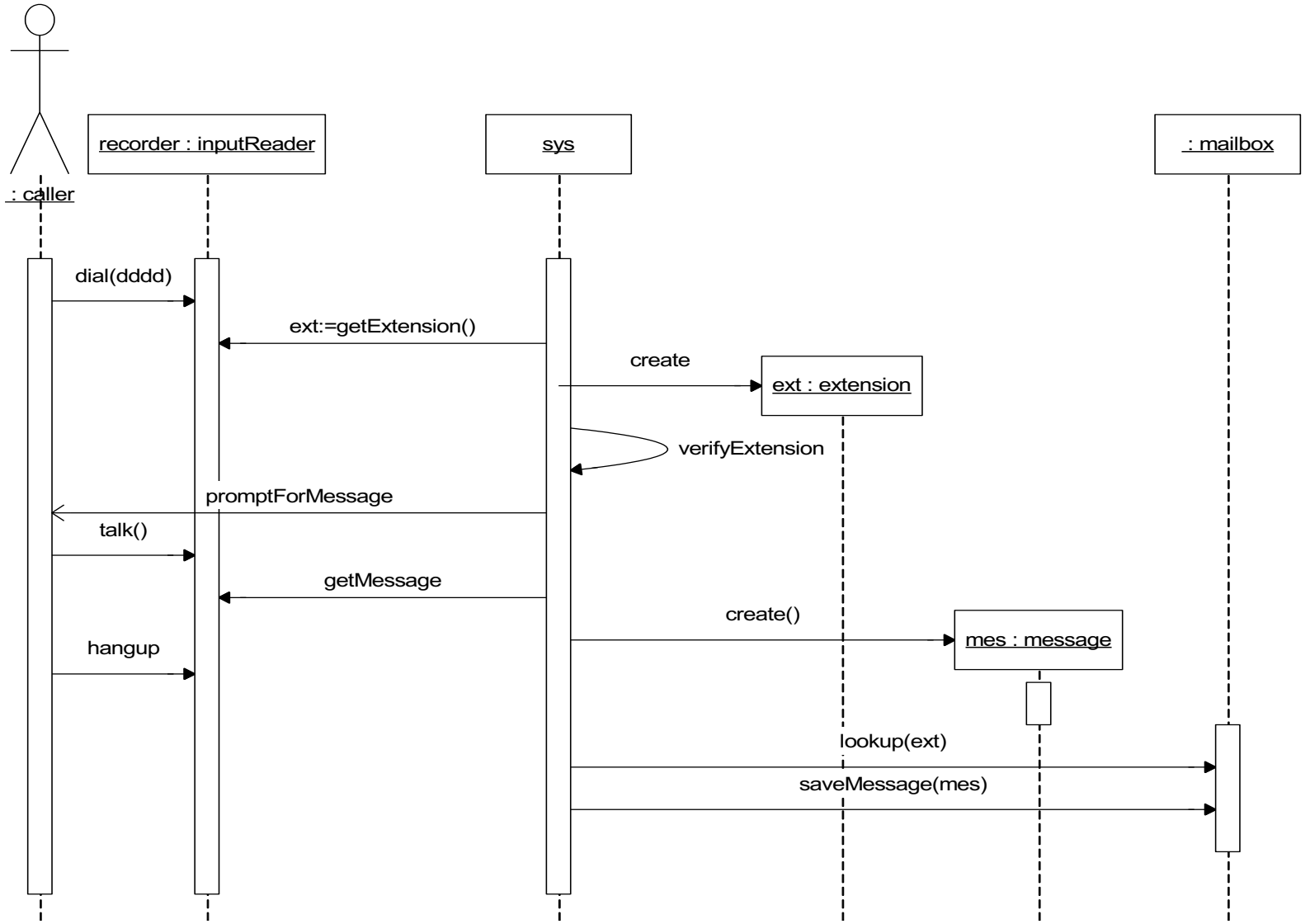


UML Sequence Diagram: Mailbox



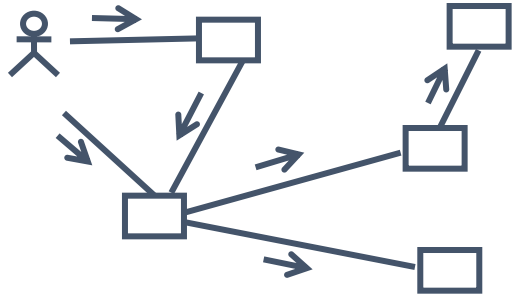
UML Sequence Diagram: Mailbox

Leave a message



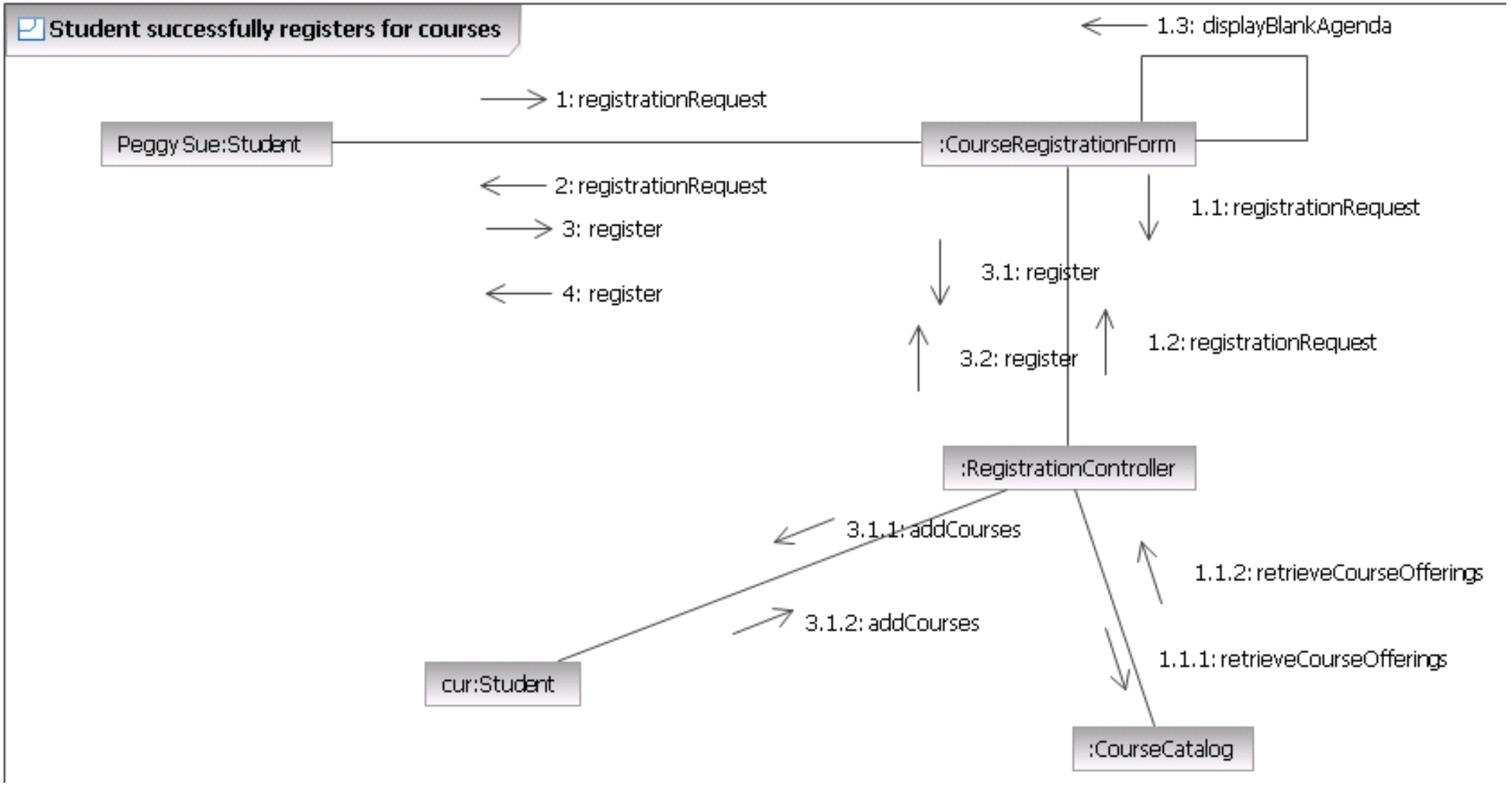
What Is a Communication Diagram?

A communication (Interaction) diagram emphasizes the organization of the objects that participate in an interaction

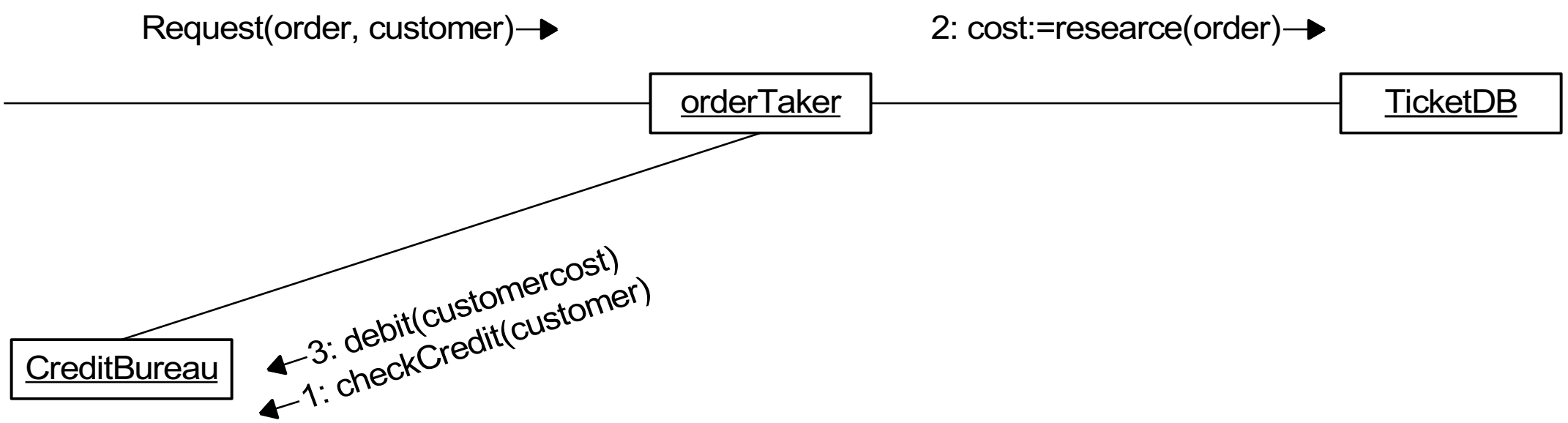


Communication Diagrams

Example


Student successfully registers for courses
1.3: displayBlankAgenda
1: registrationRequest
Peggy Sue:Student
:CourseRegistrationForm
2: registrationRequest
1.1: registrationRequest
3: register
3.1: register
4: register
3.2: register
1.2: registrationRequest
:RegistrationController
3.1.1: addCourses
1.1.2: retrieveCourseOfferings
3.1.2: addCourses
1.1.1: retrieveCourseOfferings
cur:Student
:CourseCatalog

Example Collaboration Diagram



Collaboration vs Sequence

The two diagrams really show the same information

Collaboration diagrams show more **static** structure (however, class diagrams are better at this)

Sequence diagrams clearly highlight the orderings and very useful for multi-tasking

State Diagrams

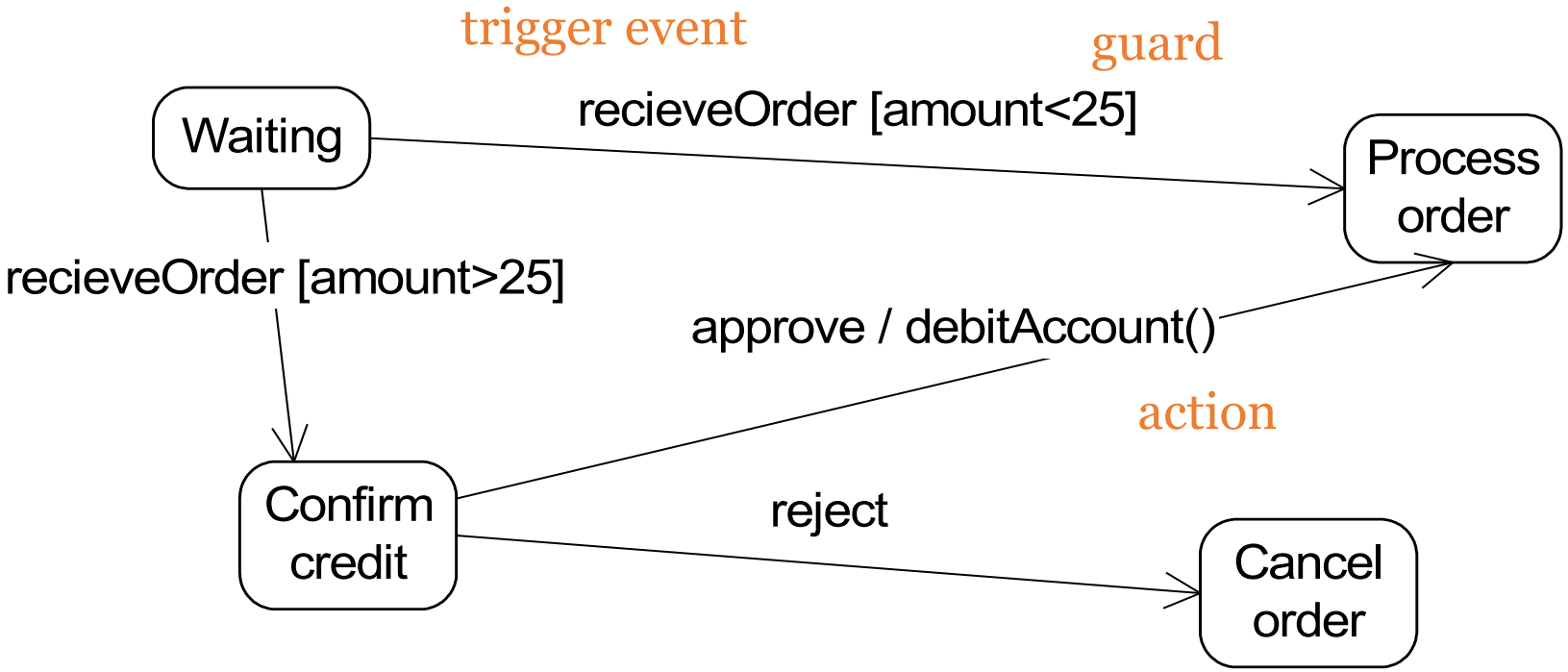
Finite state machines

Used to describe the behavior of one object (or sometimes an operator) for a number of scenarios that affect the object

They are not good for showing interaction between objects (use interaction diagrams)

Only use when the behavior of an object is complex, and more detail is needed

State Diagrams: Example

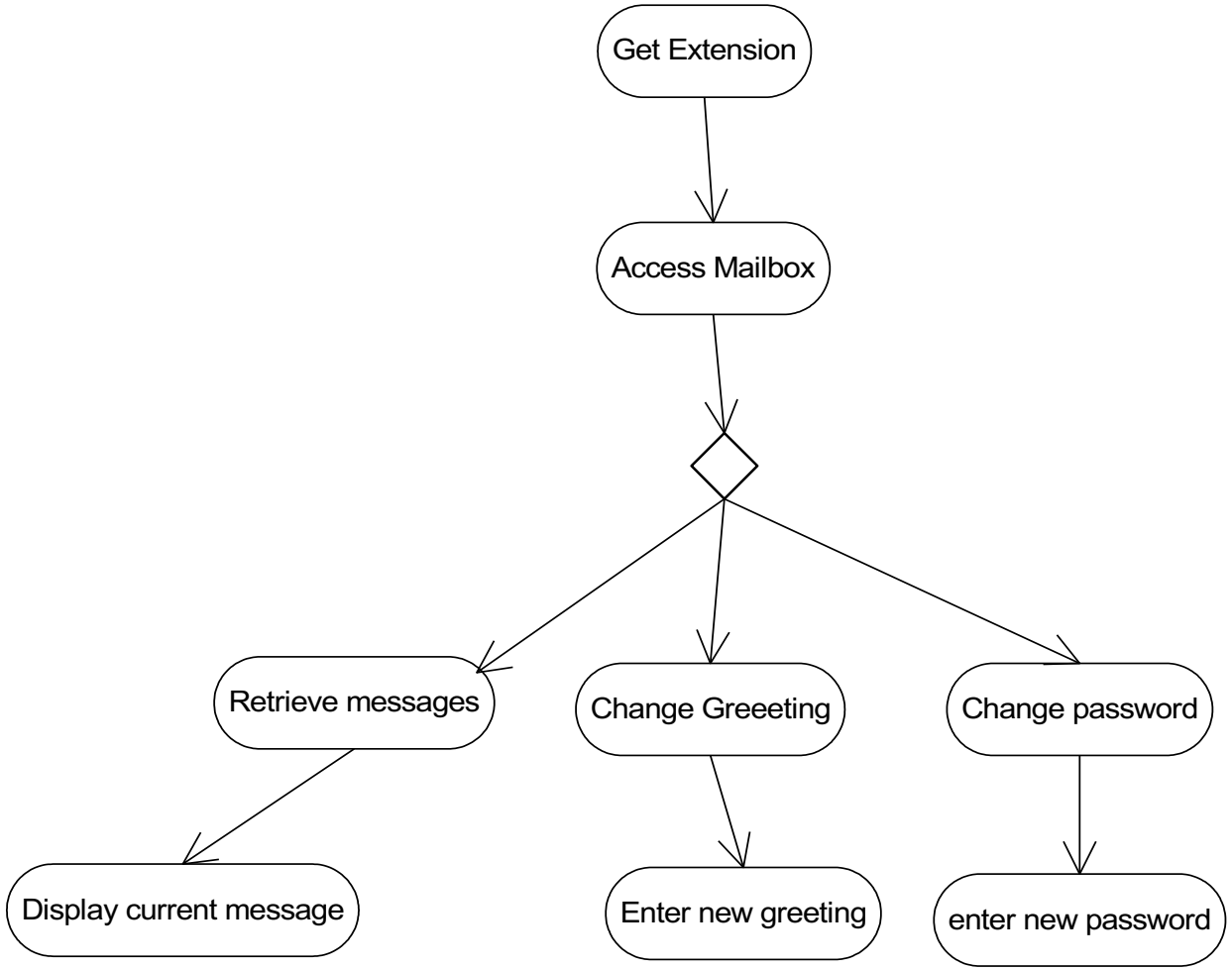


Activity Diagrams

Special form of a state machine (flow chart) – intended to model computations and workflows

States of the executing the computation not the states of an object

Example (from Mail System)



UML Part V

- **Implementation Diagrams**
- **Component diagrams**
- **Deployment diagrams**

Component Diagrams

A component is a physical entity that conforms to and realizes a set of interfaces

Bridge between logical and physical models

Can represent object libraries, COM components, Java Beans, etc.

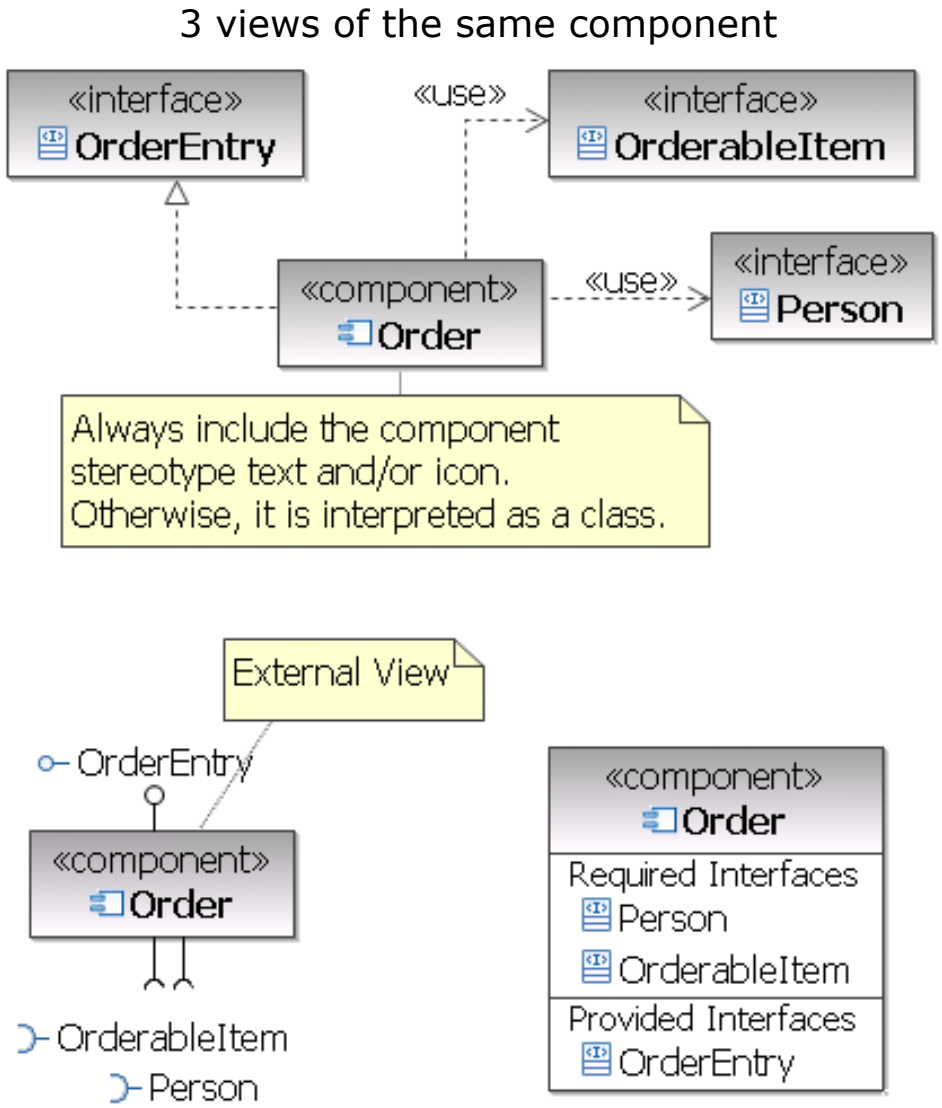
Classes represent logical abstractions; components represent physical elements that reside on a node (machine)

Components are reachable only through an interface

Components

Provide a high-level, architectural view of the system

Represent the logical components that will be running on the physical systems



Deployment Diagrams

Nodes are physical elements that represent a computational resource (machine) with association between nodes

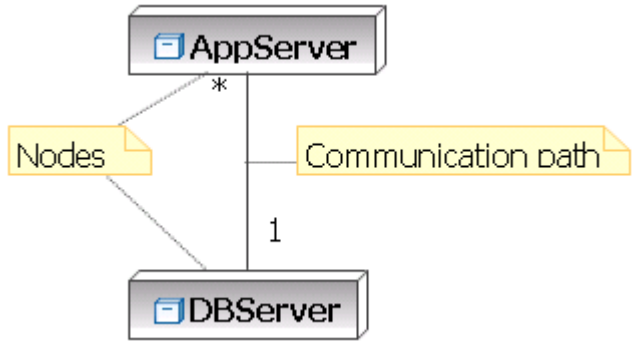
Components represent the physical packaging of logical elements

The physical deployment of components

Nodes and Communication Paths

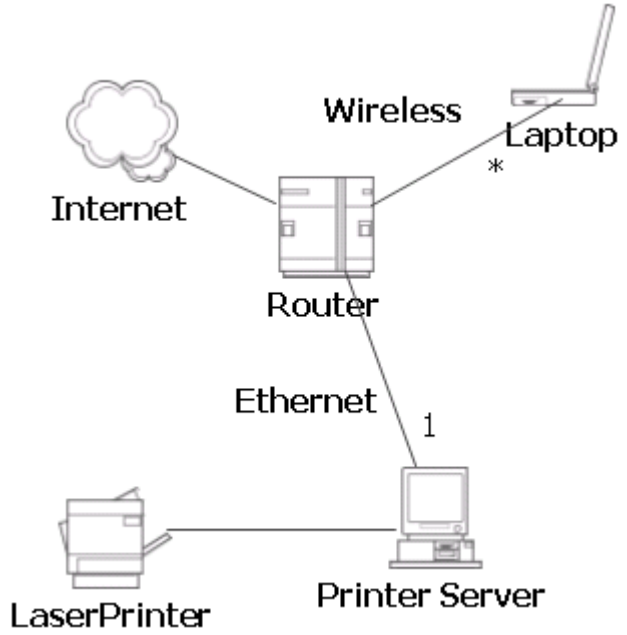
Node is computational resource upon which artifacts may be deployed for execution

- Application server, client workstation, mobile device, embedded device

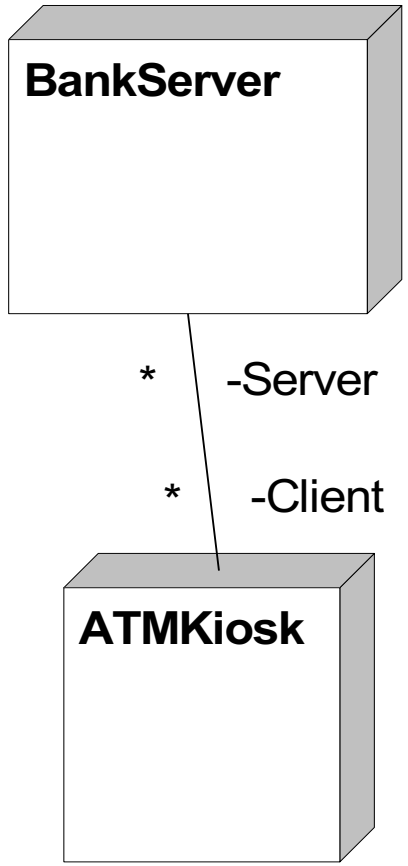


Nodes can be connected to represent a network topology by using communication paths

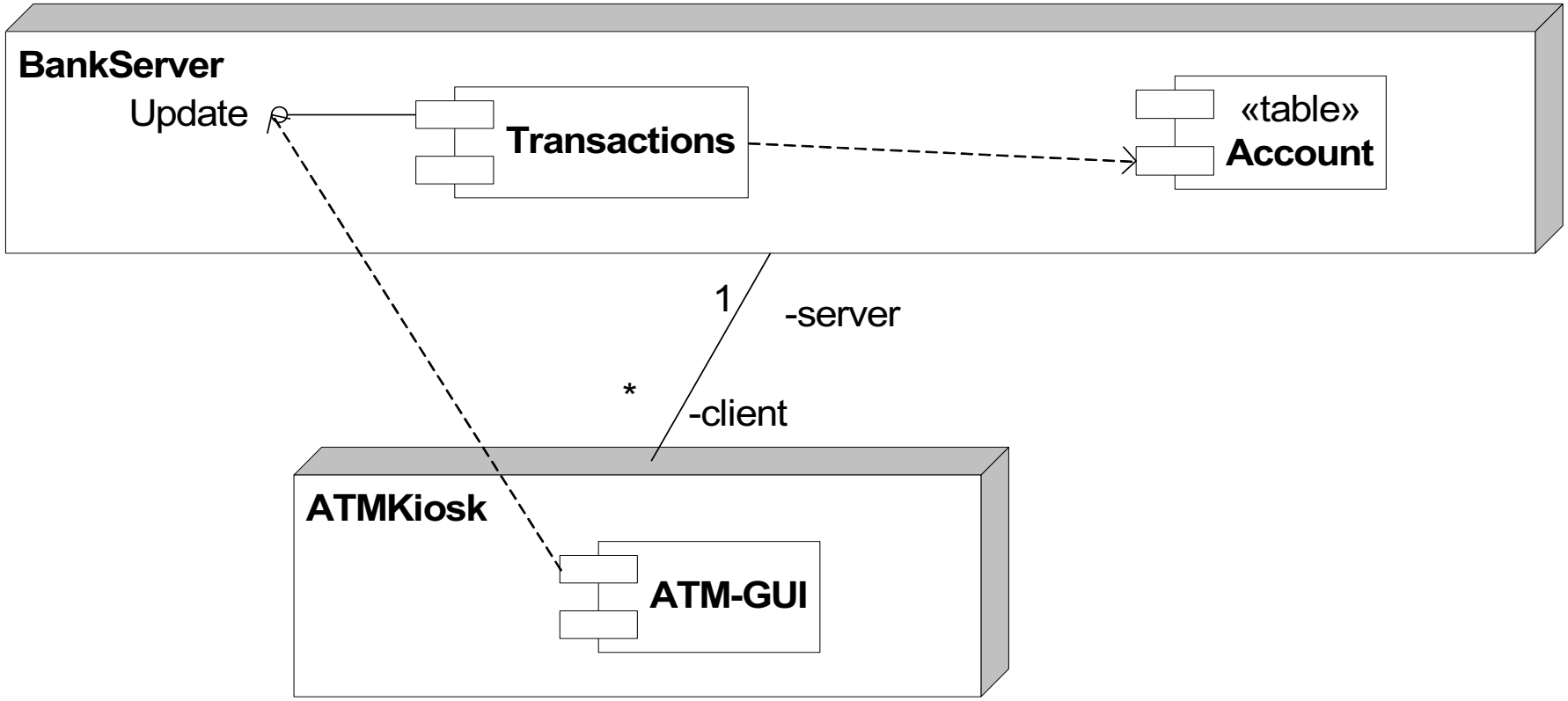
Hierarchical nodes (i.e., nodes within nodes) can be modeled using composition associations, or by defining an internal structure



Deployment Diagrams: Example



Deployment Diagrams + Components



Subsystems

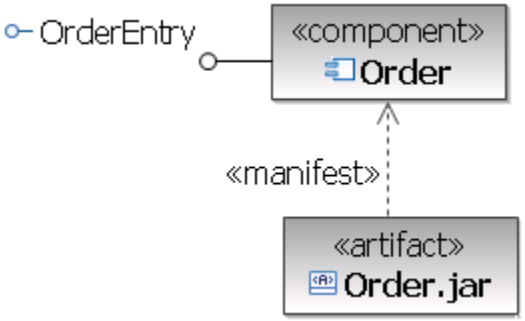
A subsystem is a specialized version of a component

Subsystems are often equated to larger components

Component stereotyped <<subsystem>>

Artifacts

A physical piece of information that is used or produced by a software development process, or by deployment and operation of a system



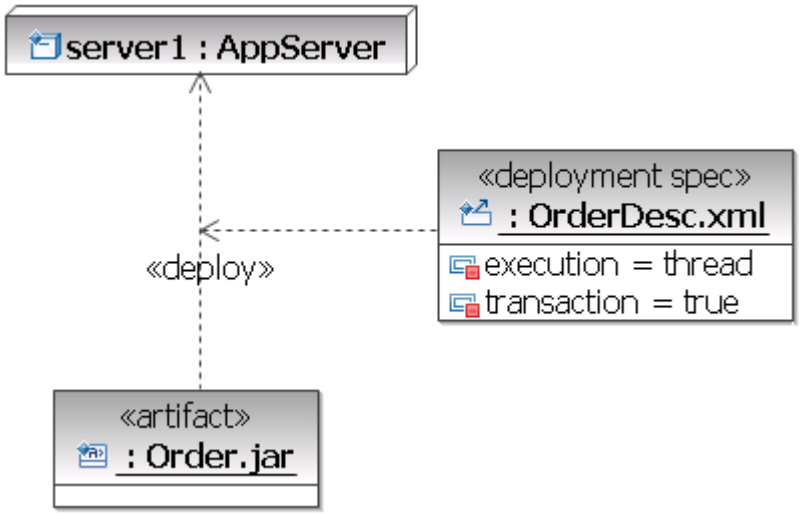
Source files, scripts, and binary executable files

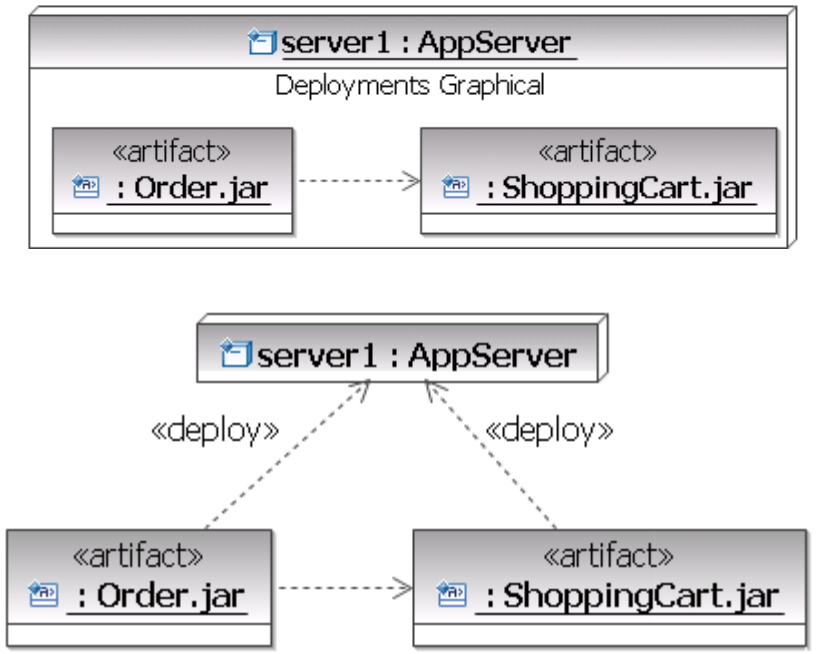
Standard UML stereotypes: <<document>>, <<executable>>, <<file>>, <<library>>, <<script>>, <<source>>

Deploying Artifacts

A deployment is the allocation of an artifact or artifact instance to a deployment target

A deployment specification can be used to specify the execution parameters of a component artifact that is deployed on a node





Two equivalent visual representations of the deployment of artifacts to a deployment target (including e dependency between the artifacts)

So Far …

- Software Crisis
- Large Software? Software Engineering
- SE is a layered technology
  - Think Quality
  - Process Waterfall vs. Agile
  - C++, UML are tools
  - Methods: **Requirements, analysis, design, implementation**, testing, release, maintenance

What is Next?

- Iterative, Evolutionary, and Agile [**Ch2** Larman,3/e]

- Case Studies [**Ch3** Larman,3/e]

- Inception [**Ch4** Larman,3/e]:
  - Inception is Not the Requirements Phase
  - Evolutionary Requirements
  - Use Cases
  - Other Requirements

Process

Iterative, Evolutionary, and Agile

Abdulkareem Alali

ACK Dale Haverstock

Based on Larman's Applying UML and Patterns Book, 3d

You should use iterative development only on projects that you want to succeed.

—Martin Fowler

Change happens slowly, then all at once

— Ernest Hemingway

Grady Booch Says …

People are more important than any process

Good people with a good process will outperform good people with no process any time

Iterative Development—Introduction

Iterative development is the basis of how OOAD is best practiced

Iterative and evolutionary development are the opposite of a sequential or "waterfall" approach

With an iterative approach there is:

- Early programming and testing,
- of a partial system
- in repeating cycles

Iterative Development—Introduction

Development begins before all the requirements are defined in detail

Feedback is used to clarify and improve the evolving specifications

Studies consistently show that the waterfall is strongly associated with the highest failure rates for software projects

Research shows that iterative methods have higher success, and productivity rates, and lower defect levels

The Unified Process (Agile UP)

A software development **process** describes an approach to building, deploying, and maintaining, software

There are several agile/iterative/evolutionary processes

UP promotes widely recognized best practices

UP is flexible and open and encourages including good practices from other iterative methods

UP is common industry professionals, you should know about it

What is Iterative Development?

A series of short, fixed-length (for example, three-week) mini-projects

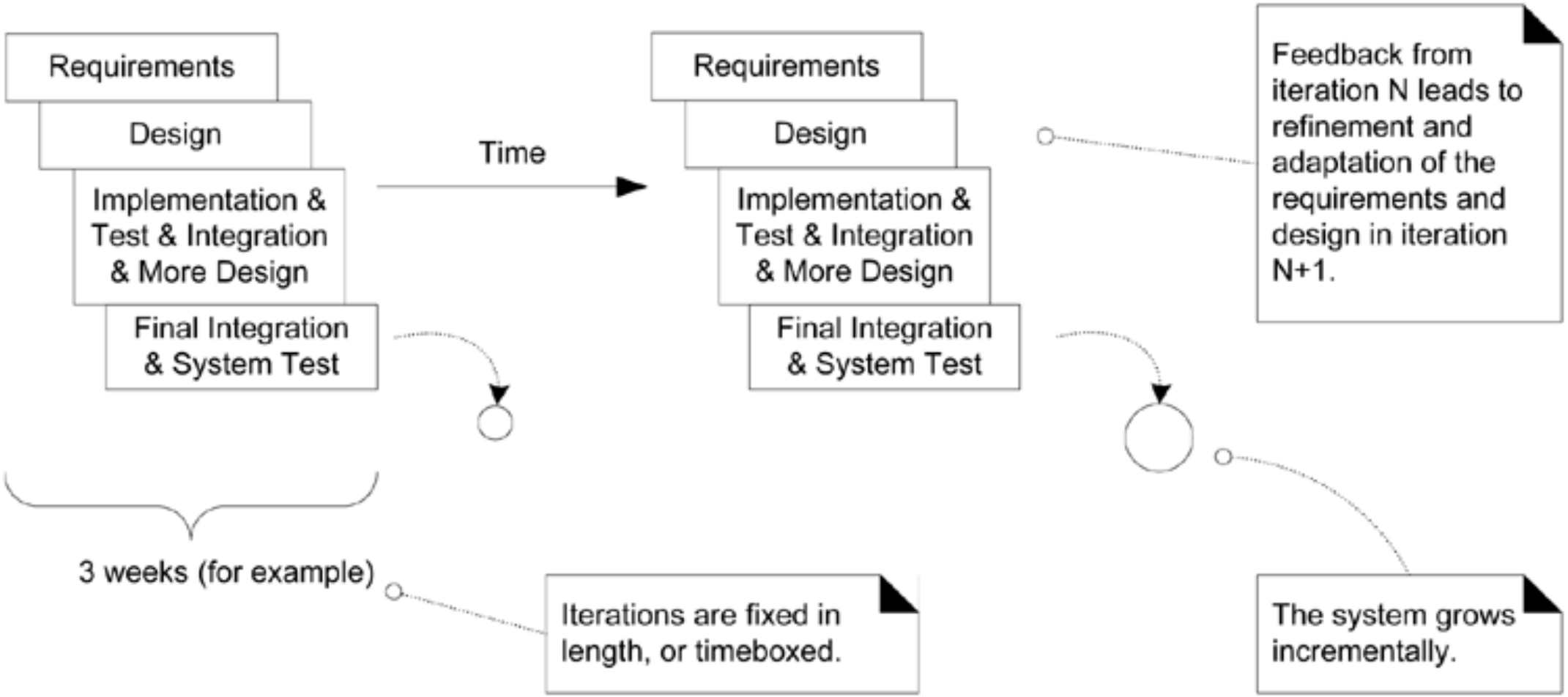
Each mini-project is called an iteration

The outcome of each iteration is a tested, integrated, and executable partial system

What happens in each Iteration?

Each iteration has it all: requirements, analysis, design, implementation, and testing activities

The system grows with each iteration and due to feedback from clients it can converge on the desired system


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.

Iterative Development— Comments

No rush to code

No long-drawn-out design step to perfect all the details before programming

Half or full day of design involving UML sketching in pairs at whiteboards

Number of iterations for the system to be usable, more iterations to a releasable

Example of Iteration Activities

Three-week iteration early in the project might have:

- One hour Monday morning team meeting clarifying the tasks and goals of the iteration
- The remainder of Monday is spent on modeling, sketching rough UML diagrams, and writing some pseudocode and design notes
- The following days are spent on implementation, testing, further design, integration, and daily builds

Iterative Development —Stakeholders

End-users see a partial system and comment on it,
Stakeholders can give feedback

There will also be demonstrations and evaluations with stakeholders, and planning for the next iteration

Corrections can be made early, rather than much later. Why?

Iterative Development —Client Issues

It is easy for, clients to:

- Not state requirements clearly
- Clients to not know exactly what they want
- Developers to misunderstand what the client wants

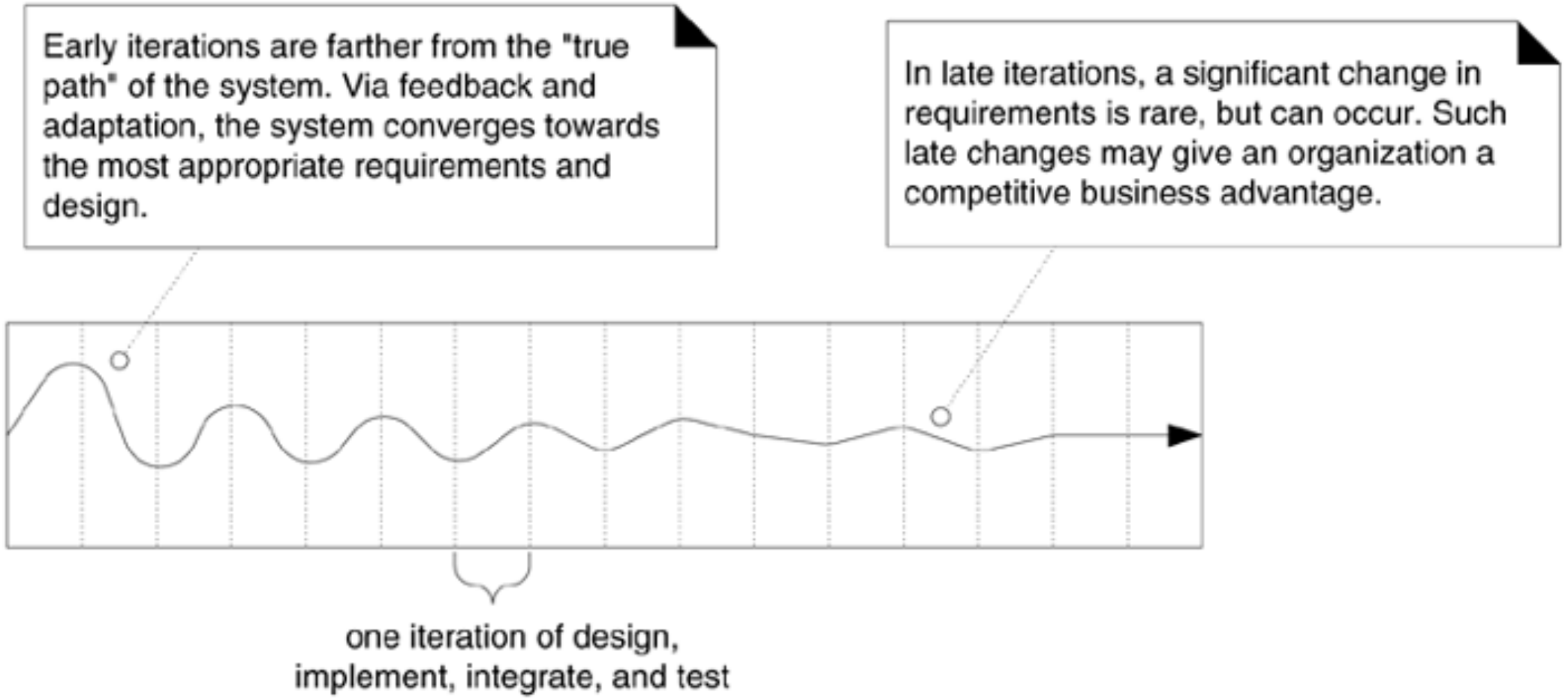
Iterative Development —Focus

Resolve **risky/critical** design decisions early rather than late

Balance of an agreed-upon set of requirements with the reality of changing requirements as stakeholders clarify their vision or the marketplace changes

As time progresses the requirements and design instability become lower

Iterative Feedback and evolution leads towards the desired system



Iterative Development —In Short

1. Less project failure
2. Better productivity
3. Lower defect rates
4. Early visible progress
5. Early handling of high-risk issues
6. Early feedback resulting in a system that more closely meets the real needs of the stakeholders
7. Managed complexity, the team is not overwhelmed by **"analysis paralysis"** or very long and complex steps
8. The **learning** within an iteration can be used to improve the development process itself as iterations progress

How Long is an Iteration?

Most methods recommend an iteration length from two and six weeks

Long iterations work against the benefits of Iterative Development

Iterations are time-boxed, fixed duration

If requirements cannot be met?

Remove some requirements from the iteration

The Pragmatic Programmer [Book by Hunt & Thomas .]

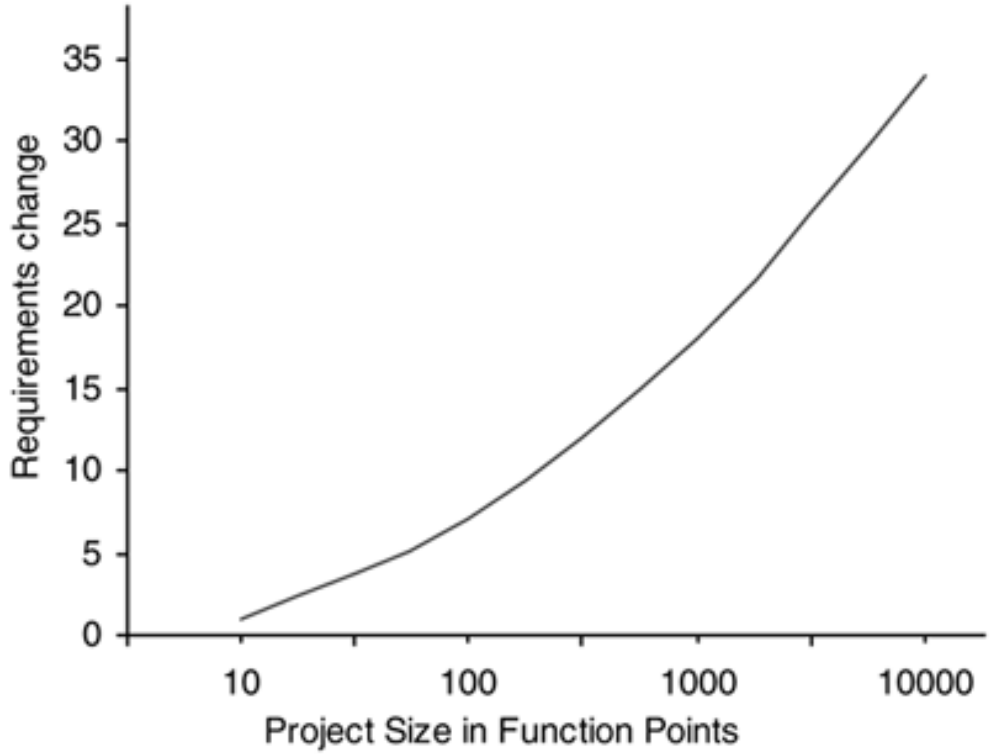
Why is the Waterfall so Failure-Prone?

The false assumption that:

- Specifications are predictable,
- Stable
- can be correctly defined at the start,
- with low change rates

Studies show that this is far from accurate

Percentage of Change on Software Projects Of Varying Sizes



Feedback is Very Important!

In complex, changing systems feedback and adaptation are key to success:

1. **Programmers** read specifications and demo to clients, clients have feedback, refine the requirements

2. From testing as **developers refine the design and models**

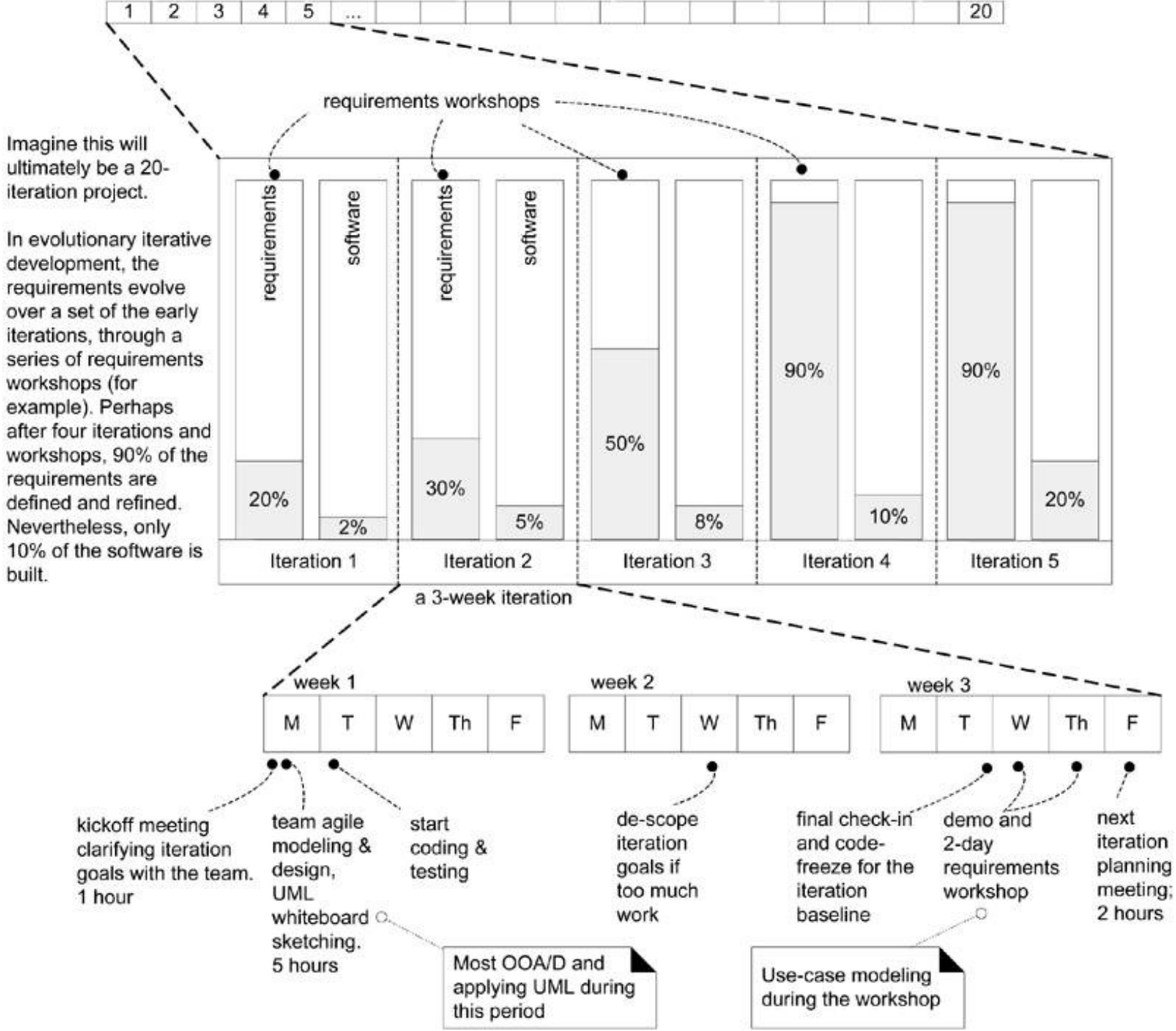
3. From the **client** to **reprioritize** features to work on in the next iteration

Iterative Planning is Risk-Driven and Client-Driven

Risk-driven and client-driven iterative planning are usually employed:

- **Difficult and risky choices first, shape the system up**
- Build **visible** features that the client cares most about

Early iterations focus on building, testing, and stabilizing the core architecture because this is typically a high risk


1 2 3 4 5 ... 20
requirements workshops
Imagine this will ultimately be a 20-iteration project.
In evolutionary iterative development, the requirements evolve over a set of the early iterations, through a series of requirements workshops (for example). Perhaps after four iterations and workshops, 90% of the requirements are defined and refined. Nevertheless, only 10% of the software is built.
requirements
software
20%
2%
30%
5%
50%
8%
90%
10%
90%
20%
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
a 3-week iteration
week 1
week 2
week 3
M T W Th F
kickoff meeting clarifying iteration goals with the team. 1 hour
team agile modeling & design, UML whiteboard sketching. 5 hours
start coding & testing
Most OOA/D and applying UML during this period
de-scope iteration goals if too much work
final check-in and code-freeze for the iteration baseline
demo and 2-day requirements workshop
Use-case modeling during the workshop
next iteration planning meeting; 2 hours

Agile Methods

Agile development employs methods that aid in **rapid** and **flexible response to change**

Manifesto for Agile Software Development:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

agilemanifesto.org

The Agile Manifesto 12 .

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation

The Agile Manifesto 12 .

6. Working software is the primary measure of progress
7. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely
9. Continuous attention to technical excellence and good design enhances agility
10. Simplicity—the art of maximizing the amount of work not done—is essential
11. The best architectures, requirements, and designs emerge from self-organizing teams
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly

Agile Modeling + UML

- Modeling (including UML) is primarily to **understand, not to document**

- **Agile modeling** has many guidelines including:
  - Don't model alone, model in pairs (or threes) at the whiteboard
  - Stick to simple, frequently used UML elements
  - Know that all **models** will be **inaccurate**, the final code or design different-sometimes dramatically different-than the model
  - …

Agile? Lean?

- Approach to **managing** and **working** on projects
- Manage complexity, constant change, and uncertainty
- **Agile (scientific method!) *versus* heavy upfront planning (waterfall) approach (We fall trap into idealism!)**
- It is **Adaptive** approach that focuses on **value** and **quality**

Agile is Based on the Scientific Method

The scientific method

1. Create a hypothesis
2. Build an experiment to test
3. Observe/learn from the results
4. Refine/repeat/iterate

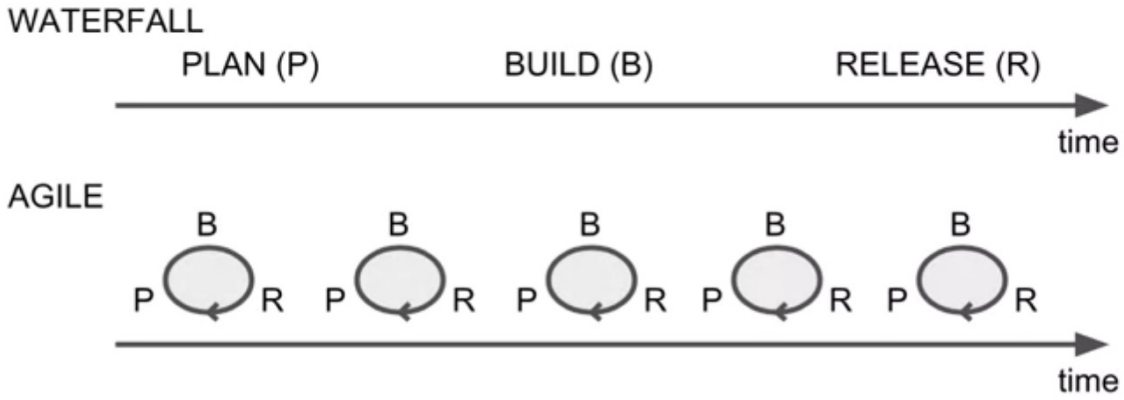
Empirical approach for **problem solving**

The agile method

1. Plan the work
2. Do the work and experiment
3. Observe/Learn from feedback
4. Refine/repeat/iterate

Experimental evidence-based methods of a **project development** leveraging knowledge from previous cycles

Waterfall



- One cycle
- Big upfront plan is wrong
  - **Can’t predict the future**
  - High value features are unknown without using it (**Waste**)
  - **Complex** and more **time** to finish
  - **Market will change** or team changes before it is done
  - **Change is expensive**
  - A lot of **obsolete** documents
  - **Feedback** is delayed

Agile is …

- **Incremental**— Plan, dev., release
- **Iterative**— Continuous improvements, *feedback*
- **Focus on value**— Start with highest value, then next
- **Empowered team**— Not a command/control system but a distributed brain, all have current knowledge, to make good quick decisions

Agile, benefits?

Customer—

Desirable (less waste, less risk) product *via* **feedback**

Value (top feature focus)

Constant **deliverables**

Response to **Change** (Business, etc.) + Knowledgeability across team

≡ **Quality**

For the team—

Empowerment (Decision making)

≡ **Satisfaction** & **Sense purpose**

Constant Learning + Questioning + **Experimentation**

≡ **Innovation** & **Entrepreneurial** spirit

ZOOM OUT—
Software Development (Pressman)

- Software Engineering is a **layered technology**
- Any engineering approach must rest on an organizational commitment to **Quality**, the bedrock that supports software engineering is a Quality *focus*
- How? ***Principles***

A Process via Applying Principles

Principles are more important than practices or tools, they tend to be foundational and long-lived while practices and tools change

"With a better tool, we can get wonderful results. But if we use it incorrectly. The tool can make things worse." "We should not forget to always use the principles ..."

Taiichi Ohno

Toyota Production System: Beyond Large-Scale Production

Lean Principles

1. Empower the team

- Diverse skills
- Trust team decisions
- Satisfaction
- Teamwork over individual craftsmanship

2. Visualize work

- Kanban board (work is visualized)
- Andon board (problems are highlighted)

3. Experiment using the scientific method

- Continuous learn and Consistent improvement
- "*Never satisfied with existing solutions*" .. *TO*
- Embraced change, "*change will not be felt as change*" .. *TO*

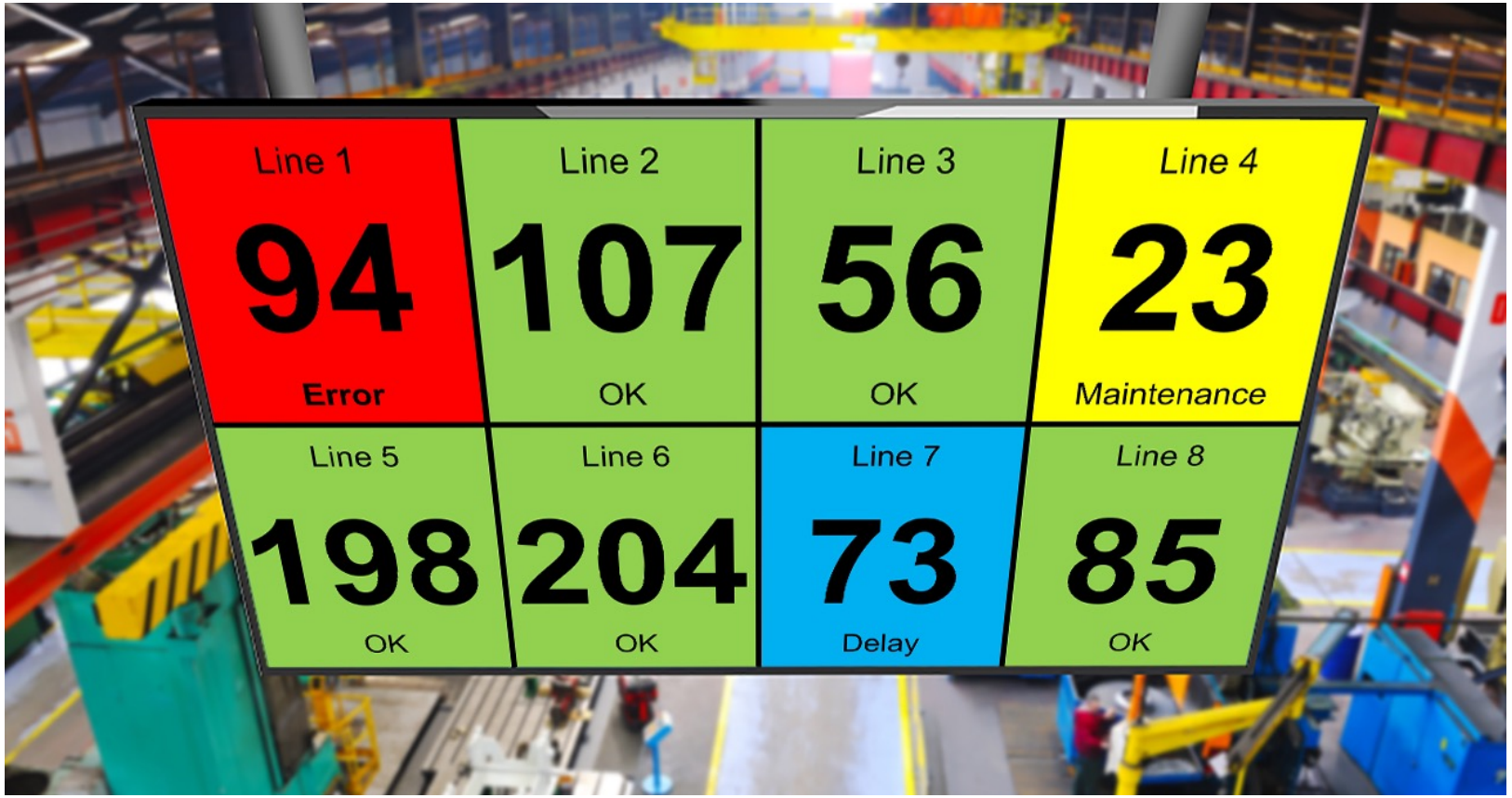
4. Improve the "flow" of value

- Whole system improvement, efficiency at each step
- Map value stream
- Limit work in progress
- Pull work
- Eliminate waste
- Reduce setup time
- Automate what should be

5. Build quality in

- Identify problems
- Fix problems once discovered, tackle root causes

Andon board —Example



Agile + Lean Principles (MishMash)

1. Empower the team
   - Build around motivated individuals (ap5)
   - Trust and environment (ap5)
   - Collaborate to create shared understanding (ap4)
   - Self-organizing teams (ap11)
2. Visualize work
   - Kanban & Andon boards. Work visualized, problems highlighted
3. Experimenting using the scientific method
   - Continuously learn and improve
   - Continuously inspect and adapt (ap12) (daily standups and retro)
   - Embrace change (more competitive) (ap2)
4. Plan, develop, deliver incrementally and frequently (ap1, ap3)
   - Prefer conversations for conveying information (ap6)
   - Collect fast/frequent feedback from customer (ap4)
   - Focus on value to satisfy customers (ap10)
   - Progress is measured by completed work items (ap7)
   - Maintain a sustainable pace (ap8)
5. Improve the "flow" of value
   - Whole system improvement, efficiency at each step
   - Map value stream .
   - Limit work in progress
   - Pull work
   - Reduce setup time
   - Automate what should be
   - Continuously strive for simplicity (ap10)
   - Eliminate waste (ap10)
6. Build quality in
   - Don't compromise on quality
   - Continuously refactor to maintain agility via simplifying & enhance quality to improve adaptability (ap9)
   - Identify problems
   - Fix problems once discovered, tackle root causes

Agile UP —In Short .

1. **Architecture-Centric/Risk and Client Driven**
2. **Use-Case driven**
3. **Iterative and Incremental** Development Process
4. **Leverages UML**
5. **Extensible/Customized** framework for specific projects

UP Main Vocabulary Terms

UP Phases

UP Disciplines

UP Artifacts (Models)

UP Phases

Inception

Approximate vision, business case, scope, vague estimates

Elaboration

Refined vision, iterative implementation of the core architecture, resolution of high risks, identification of most requirements and scope, more realistic estimates

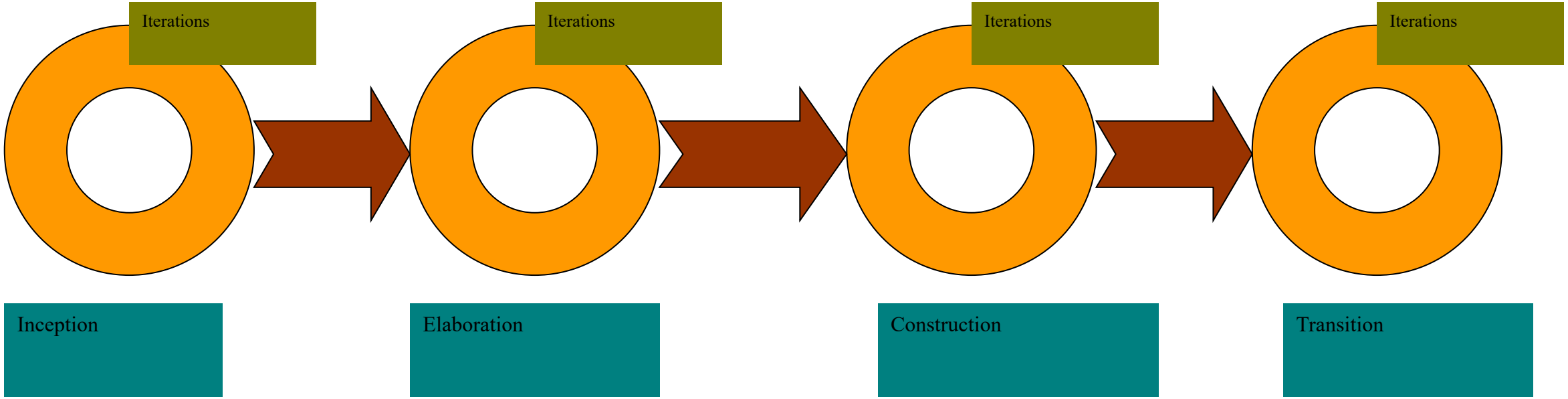
Construction

Iterative implementation of the remaining lower risk and easier elements, and preparation for deployment

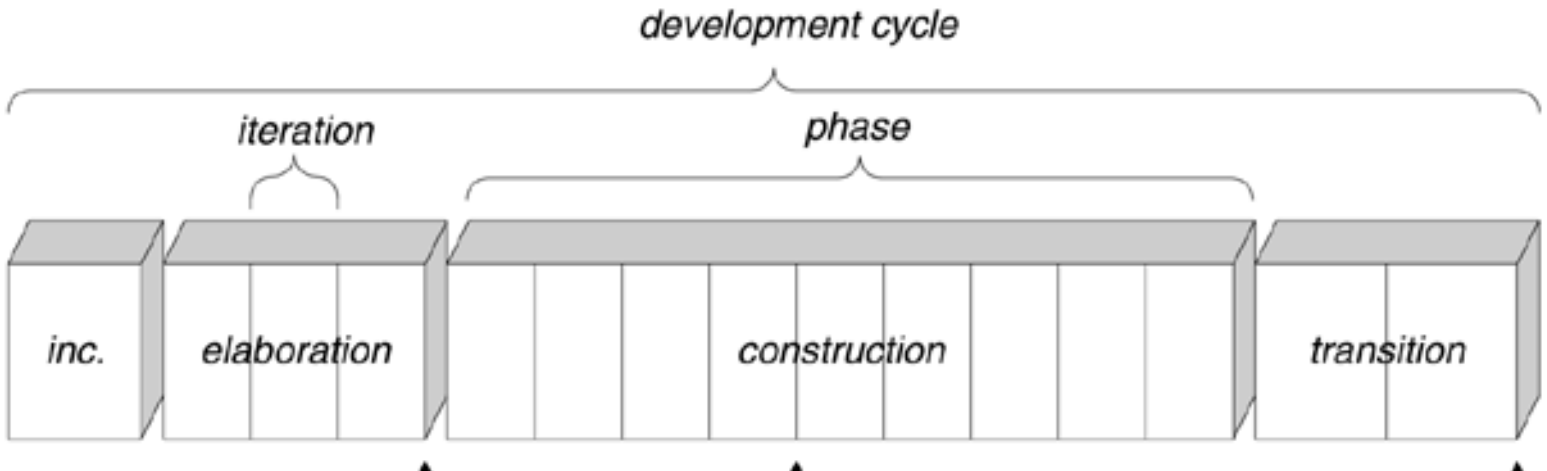
Transition

Beta tests, deployment

UP Phases
—Not a Good Chart!



UP Phases —Slightly Better One!



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.

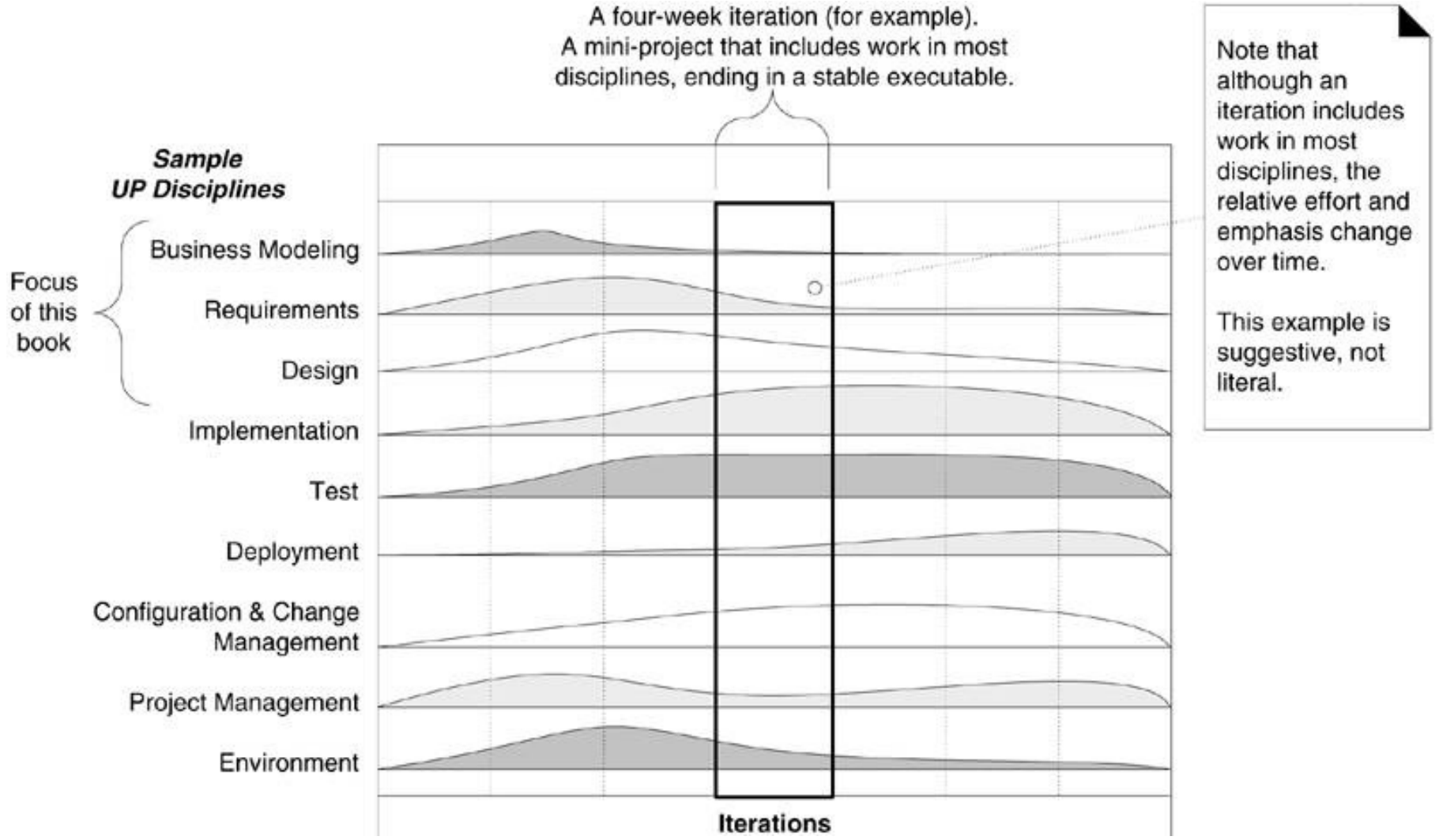
UP Disciplines
–Development Activities

- **Requirements**
  The Use-Case Model and Supplementary Specification artifacts to capture functional and non-functional requirements

- **Business Modeling (Analysis)**
  The Domain Model artifact, to visualize noteworthy concepts in the application domain

- **Design**
  The Design Model artifact, to design the software objects
- ….

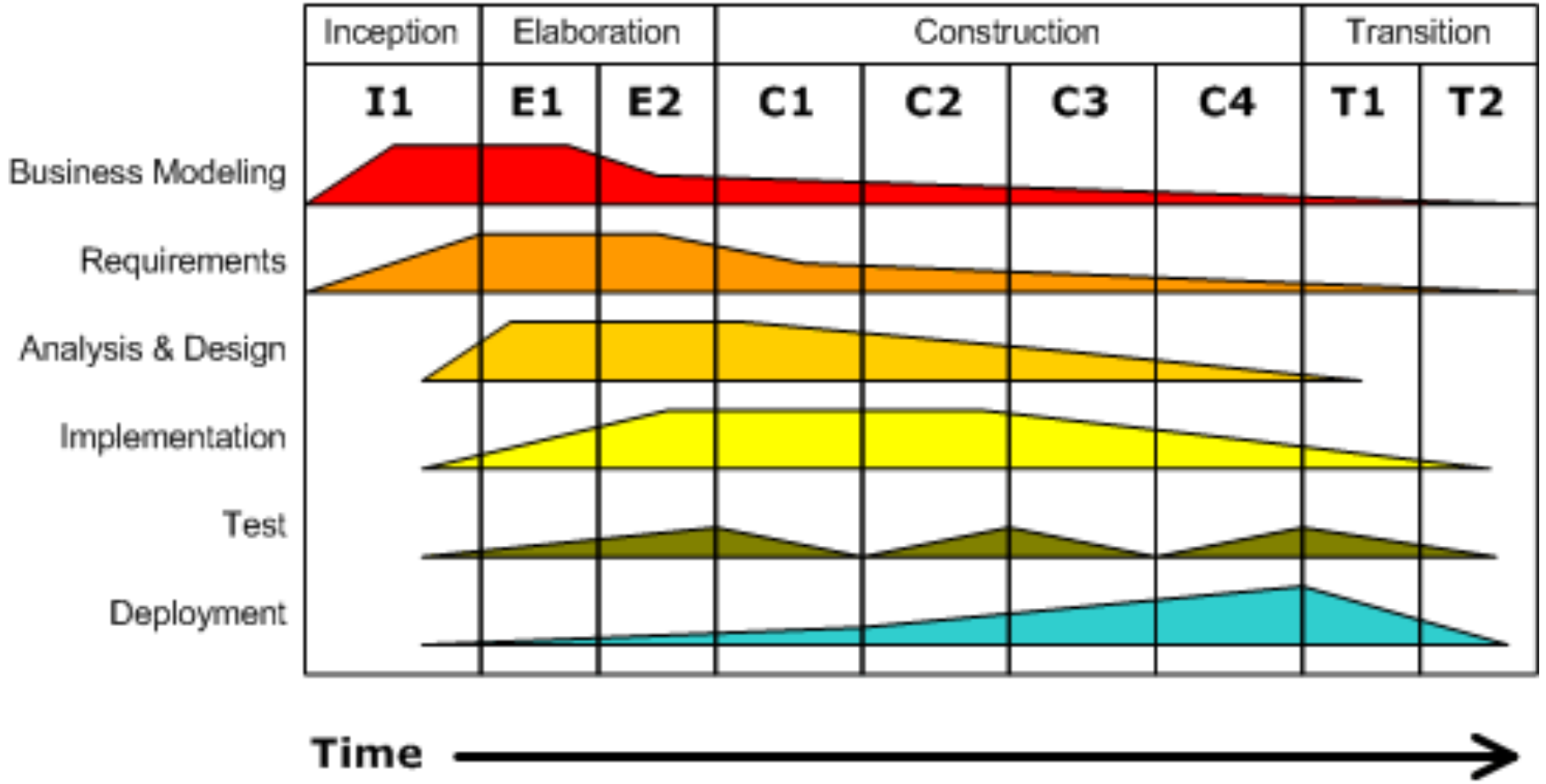
UP Disciplines Cross Iterations —Way Better!



UP Disciplines Cross Phases —The Best!

Iterative Development

Business value is delivered incrementally in time-boxed cross-discipline iterations.



UP Artifacts

- Unified Process artifacts from **requirements**:
- **Use case model** (use cases, use case diagrams)
- **Vision**
- **Supplementary specification** (nonfunctional requirements)
- **Glossary**

- More in this when Inception is discussed.

An Non Iterative/Agile Detected!

- Most of the requirements are being defined before design or implementation
- Most of the design is being done before starting implementation
- Days or weeks are spent in UML modeling before programming
- Thinking that inception is requirements, elaboration is design, and construction is implementation (the waterfall is being superimposed the UP)
- The iteration length is three months long
- Projecting in detail from start to finish via predicting all the iterations, and what should happen in each one

Case Studies

Abdulkareem Alali

ACK Dale Haverstock

Based on Larman's Applying UML and Patterns Book, 3d

Few things are harder to put up with than a good example

—Mark Twain

Case Studies, Larman E/3 Uses A Couple Of Case Studies

Most applications are comprised of these **layers**:

- Core application **logic**, **UI** components, **Database access**, Other **external resource** access
- Case studies focus on **OOAD** is in the core application logic layer, UI and other layers are often technology/platform dependent
- OOAD skills learned by focusing on the application logic layer are applicable to all other layers or components
- The two case studies will have an iterative OOAD approach applied to them

Case Study 1 —POS System

- First case study is called the NextGen point-of-sale (POS) system
- A computer application and hardware used to record sales and handle payments
- It's used in retail stores, restaurants, etc.
- Provides interesting requirement and design problems

A POS System MUSTs

- Must interface to various external applications, such as a tax calculator systems and inventory control
- Must be relatively fault-tolerant
- Must support multiple client-side terminals and interfaces
- Must be flexible in the support of varying business rules

Case Study 2 —Monopoly Game

- Monopoly will provide a contrasting problem domain

- Software version of the game will run with real and simulated players