

Software Engineering (CSPE41)

Summer 2023

Lecture 1:
Software and Software Engineering Fundamentals

Instructor: C. Oswald

Slides based on various web sources including Pressman's text

What is Software?

Software is:

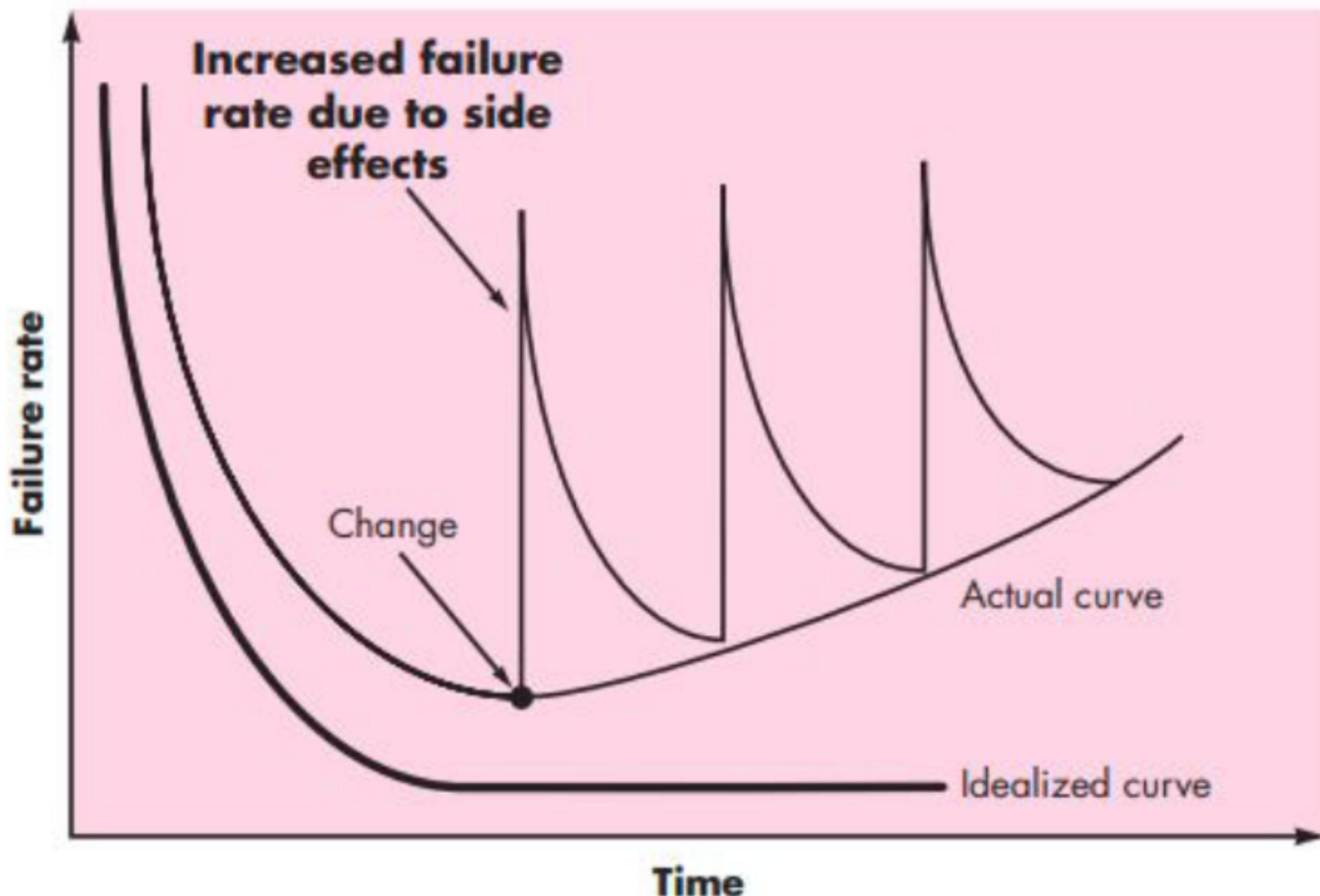
- **instructions** (computer programs) that when executed provide desired features, function, and performance;
- data **structures** that enable the programs to adequately manipulate information and
- **documentation** that describes the operation and use of the programs. (Pressman, 2017)

*Software is a set (configuration) of computer **programs** and associated **documentation** and data.
(Sommerville, 2001)*

Software Characteristics

- *Software is developed or engineered, it is **not** manufactured in the classical sense.*
- *Software doesn't "wear out."*
- *Although the industry is moving toward component-based construction, most software continues to be custom-built.*

Wear vs. Deterioration



Software Applications

- System software
- Application software
- Engineering/scientific software
- Embedded software
- Product-line software
- WebApps (Web applications)
- AI software

Software Engineering

The seminal definition:

- *[Software engineering is] the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.*

Software Engineering

The Institute of Electrical and Electronic Engineers (IEEE) definition:

Software Engineering: (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1).

A Layered Technology

<https://www.mlsu.ac.in/econtents/>

16_EBOOK-7th_ed_software_engineering_a_practitioners_approach_by_roger_s._press
man_.pdf

A Process Framework

Framework Activities

- Communication
- Planning
- Modeling
 - Analysis of requirements
 - Design
- Construction
 - Code generation
 - Testing
- Deployment

The Essence of Practice

Polya suggests:

1. *Understand the problem* (communication and analysis).
2. *Plan a solution* (modeling and software design).
3. *Carry out the plan* (code generation).
4. *Examine the result for accuracy* (testing and quality assurance).

Seven General principles

- The Reason It All Exists
- KISS (Keep It Simple, Stupid!)
- Maintain the Vision
- What You Produce, Others Will Consume
- Be Open to the Future
- Plan Ahead for Reuse
- Think!

Software Myths

Myth: *We already have a book that's full of standards and procedures for building software. Won't that provide my people with everything they need to know?*

Reality: ?

Myth: *If we get behind schedule, we can add more programmers and catch up (sometimes called the "Mongolian horde" concept).*

Reality: ?

Software Myths

Myth: A general statement of objectives is sufficient to begin writing programs—we can fill in the details later.

Reality: ?

Myth: Once we write the program and get it to work, our job is done

Reality: ?

Case Study

Mentcare: A patient information system for mental health care

- A patient information system to support mental health care is a medical information system that maintains information about patients suffering from mental health problems and the treatments that they have received.
- Most mental health patients do not require dedicated hospital treatment but need to attend specialist clinics regularly where they can meet a doctor who has detailed knowledge of their problems.
- To make it easier for patients to attend, these clinics are not just run in hospitals. They may also be held in local medical practices or community centres.

Mentcare

- Mentcare is an information system that is intended for use in clinics.
- It makes use of a centralized database of patient information but has also been designed to run on a PC, so that it may be accessed and used from sites that do not have secure network connectivity.
- When the local systems have secure network access, they use patient information in the database but they can download and use local copies of patient records when they are disconnected.

Mentcare goals

- To generate management information that allows health service managers to assess performance against local and government targets.
- To provide medical staff with timely information to support the treatment of patients.

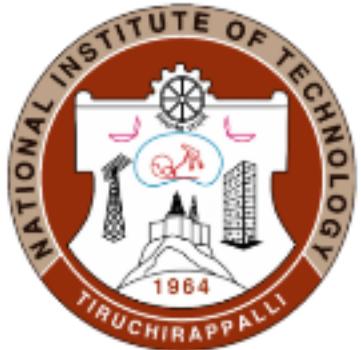
The organization of the Mentcare system

Key features of the Mentcare system

- Individual care management
- Patient monitoring
- Administrative reporting

Mentcare system concerns

- Privacy
- Safety



Software Engineering (CSPE41)

Summer 2023

Lecture 5:

Iterative and Agile Models of SDLC

Slides based on various web sources including Pressman's text

Iterative and Agile Development



Engineering software is a big job



Variety of tasks:

- Requirements
- Design
- Implementation
- Verification (testing)
- Maintenance

Practical issue:
What order should tasks be done in?
That is, what *process* to use?



Using Waterfall turns out to be a *poor practice*

- High failure rates
- Low productivity
- High defect rates



Statisti

45% **c** of features
in requirements
never used

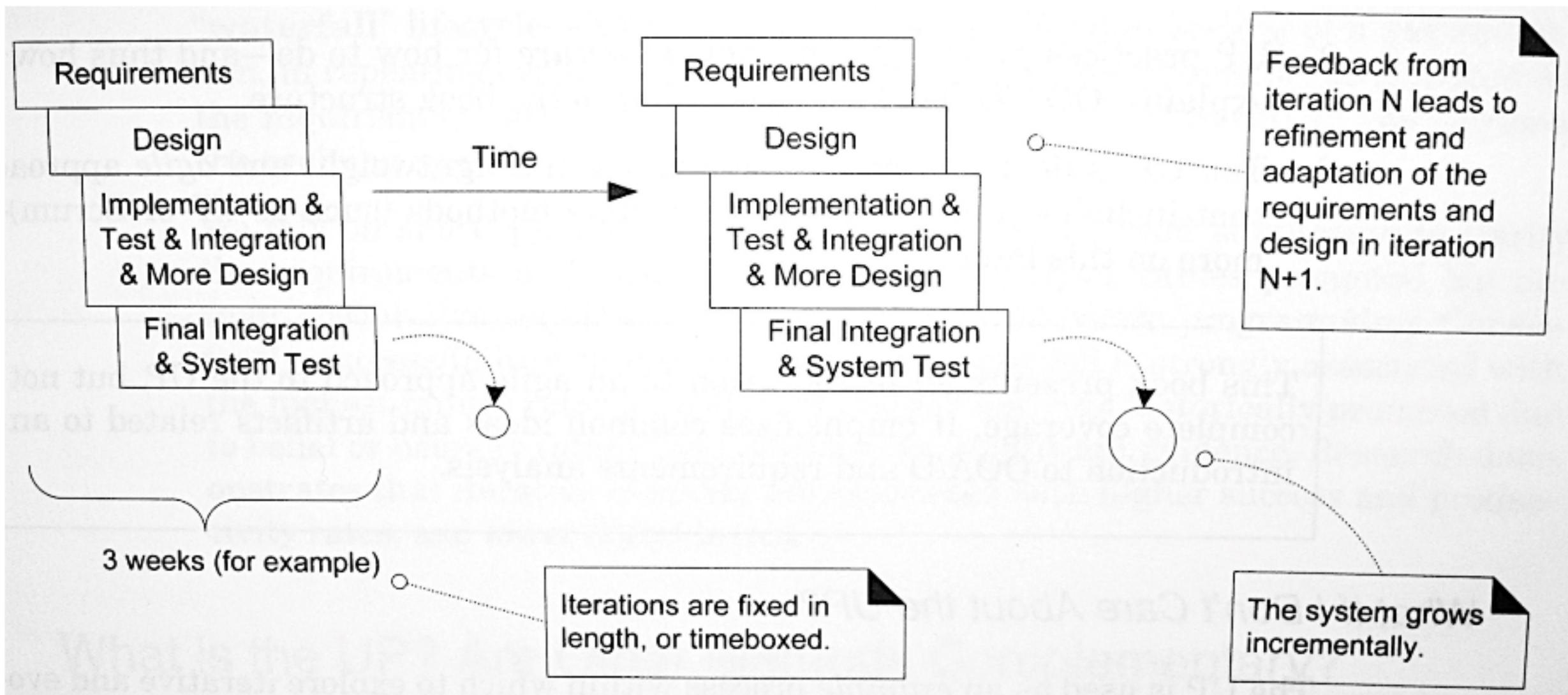


Statisti

Early **c** schedule
and estimates
off by up to 400%

Iterative and incremental development

also called *iterative and evolutionary*



Iterative, Incremental Approach



How long should iterations be?

- Short is good
- 2 to 6 weeks
- 1 is too short to get meaningful feedback
- Long iterations subvert the core motivation



Iterative and incremental development addresses the “*yes... but*” problem

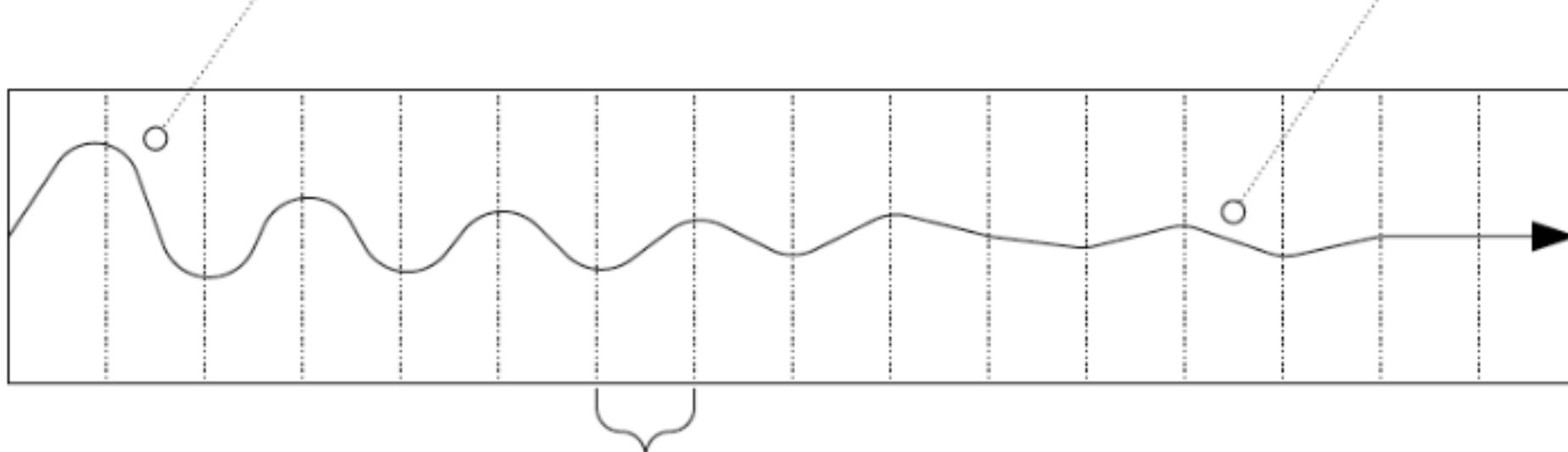


Yes, that's what I asked for,
but now that I try it, what I
really need is something
slightly different.

System converges over time

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 in requirements is rare, but can occur. Such late changes may give an organization a competitive business advantage.



one iteration of design,
implement, integrate, and test

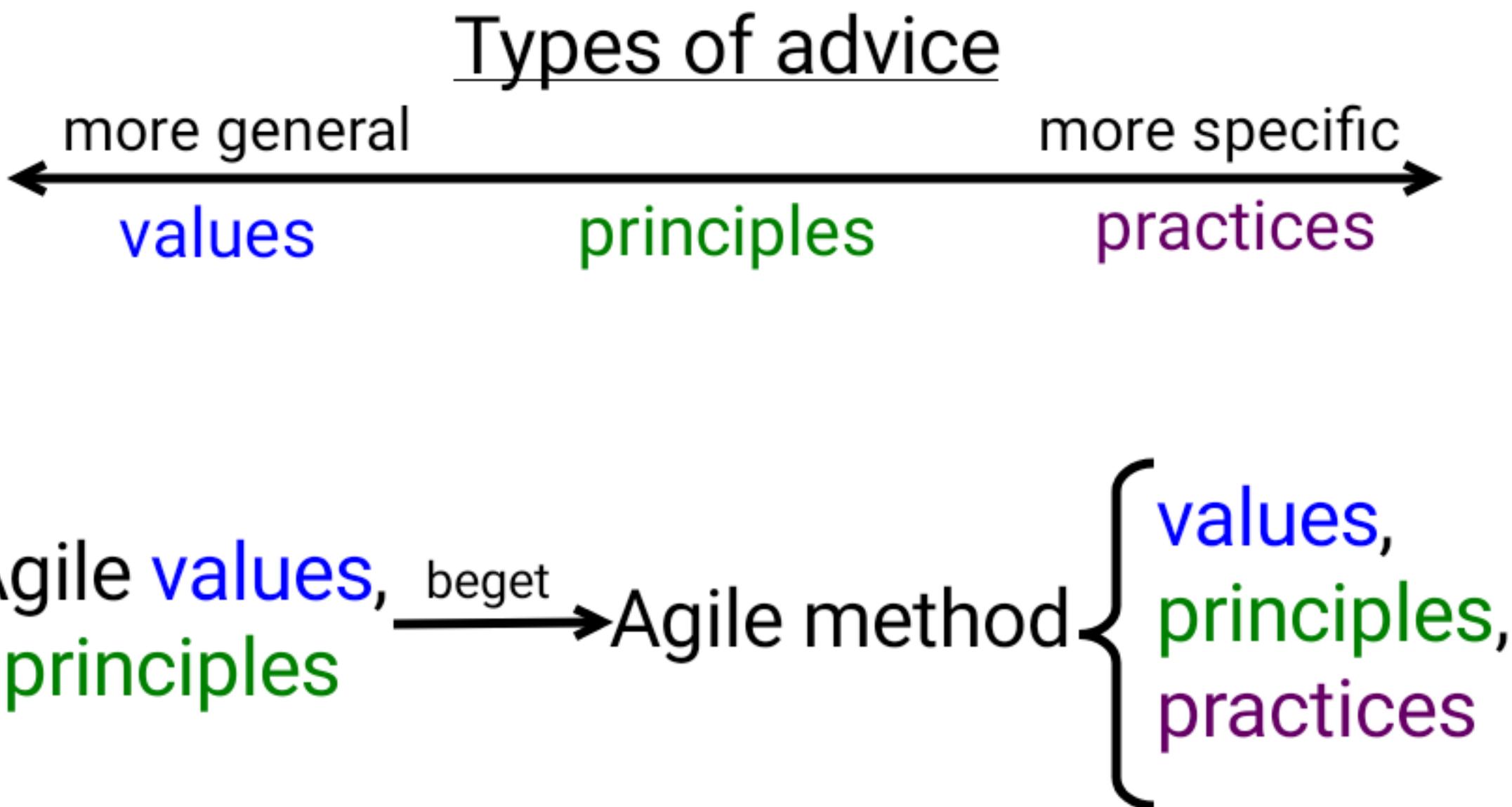
Iterative and incremental development
is a broad approach

But how to operationalize?

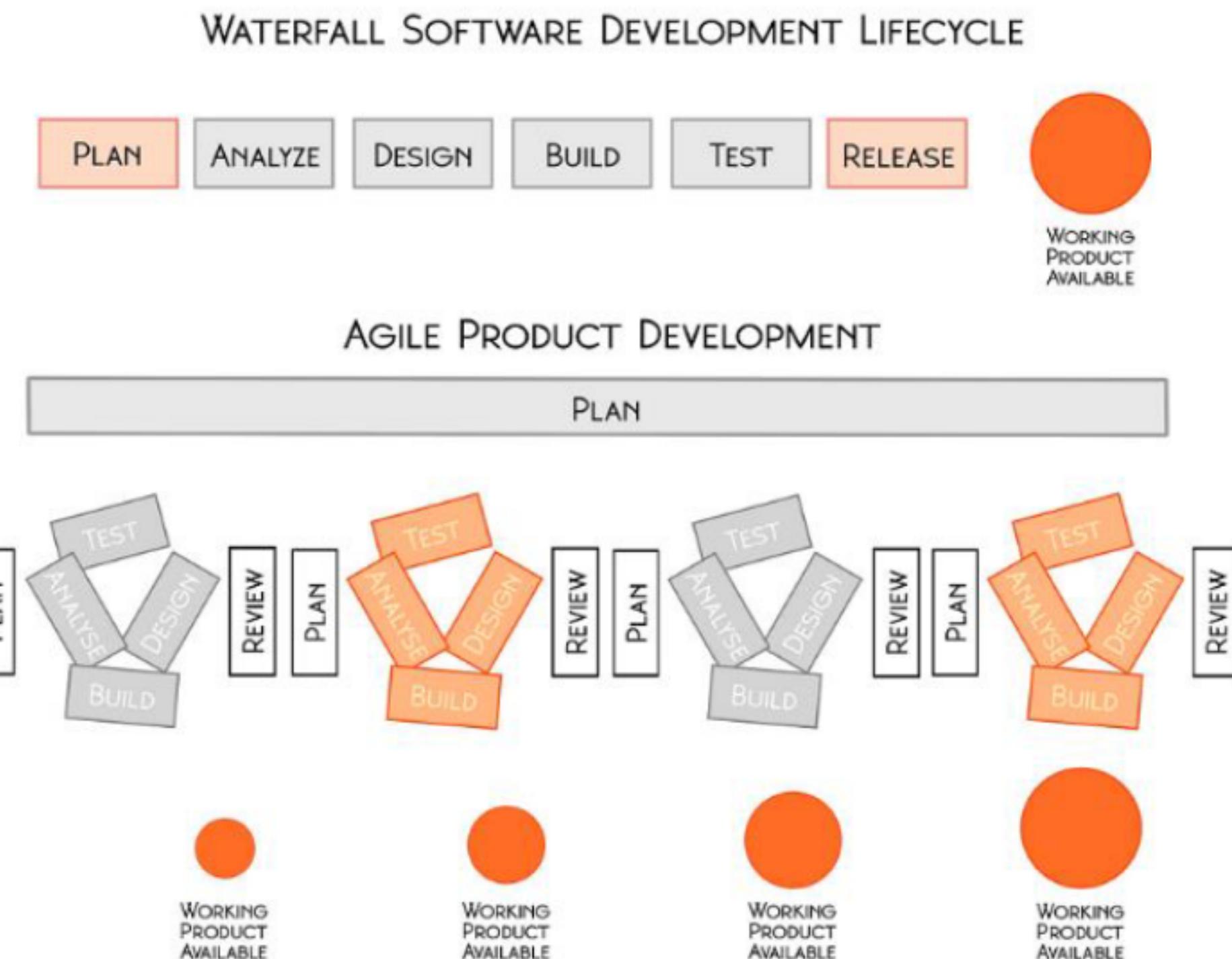
To help with that, there are
more specific methods and practices

In particular, there are *agile methods*

What is an agile method?
Where does it come from?



Waterfall Vs. Agile development plan



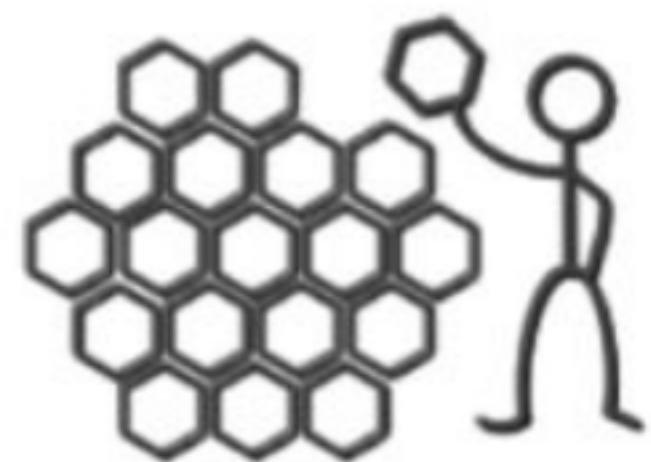
<https://www.scrum.org/resources/blog/what-iterative-incremental-delivery-hunt-perfect-example>

Waterfall Vs. Agile development plan

THE WATERFALL PROCESS



THE AGILE PROCESS



<https://www.scrum.org/resources/blog/what-iterative-incremental-delivery-hunt-perfect-example>

Agile Methods

- Dissatisfaction with the overheads involved in software design methods of the 1980s and 1990s led to the creation of agile methods. These methods:
 - Focus on the code rather than the design
 - Are based on an iterative approach to software development
 - Are intended to deliver working software quickly and evolve this quickly to meet changing requirements.
- The aim of agile methods is to reduce overheads in the software process (e.g. by limiting documentation) and to be able to respond quickly to changing requirements without excessive rework.

Agile Software Development

- A group of software development methods based on iterative and incremental development, where requirements and solutions evolve through collaboration between self-organizing, cross-functional teams.
- It promotes **adaptive planning, evolutionary development and delivery, a time-boxed iterative approach, and encourages rapid and flexible response to change.**

Agile manifesto

In 2001, Kent Beck and 16 other noted SD's writers and consultants ("Agile Alliance") stated:

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

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

That is, while there is value in the items on the right, we value the items on the left more.

Value 1: Individuals and Interactions over Processes and Tools

- **Strong players:** a must, but can fail if don't work together.
- **Strong player:** not necessarily an 'ace;' work well with others!
 - Communication and interacting is **more important** than raw talent.
- **'Right' tools** are vital to smooth functioning of a team.
- **Start small.** Find a free tool and use until you can demo you've outgrown it. Don't assume bigger is better. Start with white board; flat files before going to a huge database.
- **Building a team** more important than **building environment.**
 - Some managers build the environment and expect the team to fall together.
 - Doesn't work.
 - Let the team build the environment on the **basis of need.**

Value 2: Working Software over Comprehensive Documentation

- **Code** – not ideal medium for communicating rationale and system structure.
 - Team needs to produce human readable documents describing system and design decision rationale.
- **Too much documentation is worse than too little.**
 - Take time; more to keep in sync with code; Not kept in sync? it is a lie and misleading.
- **Short rationale and structure document.**
 - Keep this in sync; Only highest level structure in the system kept.

Value 3: Customer Collaboration over Contract Negotiation (1 of 2)

- Not possible to describe software requirements up front and leave someone else to develop it within cost and on time.
- Customers cannot just cite needs and go away
- Successful projects require **customer feedback on a regular and frequent basis** – and not dependent upon a contract or Statement of Work (SOW)

Value 3: Customer Collaboration over Contract Negotiation (2 of 2)

- Best contracts are NOT those specifying requirements, schedule and cost.
 - Become meaningless shortly.
- Far better are contracts that govern the way the development team and customer will work together.
- Key is intense collaboration with customer and a contract that governed collaboration rather than details of scope and schedule
 - Details ideally not specified in contract.
 - Rather contracts could pay when a block passed customer's acceptance tests.
 - With frequent deliverables and feedback, acceptance tests never an issue.

Value 4: Responding to Change over Following a Plan

- Our plans and the ability to respond to changes is critical!
- Course of a project cannot be predicted far into the future.
 - Too many variables; not many good ways at estimating cost.
- Tempting to create a PERT or Gantt chart for whole project.
 - This does Not give novice managers control.
 - Can track individual tasks, compare to actual dates w/planned dates and react to discrepancies.
 - But the structure of the chart will degrade
 - As developers gain knowledge of the system and as customer gains knowledge about their needs, some tasks will become unnecessary; others will be discovered and will be added to 'the list.'
 - In short, the plan will undergo changes in *shape* not just dates

Value 4: Responding to Change over Following a Plan

- Better planning strategy – make detailed plans for the next few weeks, very rough plans for the next few months, and extremely crude plans beyond that.
- Need to know what we will be working on the next few weeks; roughly for the next few months; a vague idea what system will do after a year.
- Only invest in a detailed plan for immediate tasks; once plan is made, difficult to change due to momentum and commitment.
 - But rest of plan remains flexible. The lower resolution parts of the plan can be changed with relative ease.

Iterative approach example – Web Form

Goal: to build an online application for Insurance



<https://www.scrum.org/resources/blog/what-iterative-incremental-delivery-hunt-perfect-example>

The principles of agile methods

Principle	Description
Customer involvement	Customers should be closely involved throughout the development process. Their role is provide and prioritize new system requirements and to evaluate the iterations of the system.
Incremental delivery	The software is developed in increments with the customer specifying the requirements to be included in each increment.
People not process	The skills of the development team should be recognized and exploited. Team members should be left to develop their own ways of working without prescriptive processes.
Embrace change	Expect the system requirements to change and so design the system to accommodate these changes.
Maintain simplicity	Focus on simplicity in both the software being developed and in the development process. Wherever possible, actively work to eliminate complexity from the system.

What is Agility?

Ivar Jacobson says:

- An agile team is a nimble team able to appropriately respond to changes.
- The pervasiveness of change is the primary driver for agility.
- Encourages team structures and attitudes that make
 - communication (among team members, between
 - technologists and business people, between
 - software engineers and their managers) more

12 principles of Agility (a few...)

- Our highest priority is to **satisfy** the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development.
- Build projects around **motivated** individuals.
- Face-to-face conversation

Human Factors

- As Cockburn and Highsmith state, “Agile development focuses on the talents and skills of individuals, molding the process to specific people and teams.”

The key point in this statement is

*The process molds to the needs of the people and team,
not the other way around*

Competence

Common focus

Collaboration

Mutual trust and respect – “jelled” team”

Few Agile Success @

Products...

- Azure DevOps Services (VSTS)
- Jixee (Bug and Issue Tracker)
- SauceLabs (Cloud based platform for testing)
- Gitlab
- monday.com (Work OS)

Industries...

- Philips
- Amazon
- VistaPrint
- JP Morgan Chase
- Sky

Examples of agile methods and their practices

- Scrum

- Common project workroom
- Self-organizing teams
- Daily scrum
- ...

- Extreme programming (XP)

- Pair programming
- Test-driven development
- Planning game



Some more Agile Methods

- Adaptive Software Development (ASD)
- Feature Driven Development (FDD)
- Crystal Clear
- Dynamic Software Development Method (DSDM)
- Rapid Application Development (RAD)
- Rational Unify Process (RUP)

Agile SDLC's

- Speed up or bypass one or more life cycle phases
- Usually less formal and reduced scope
- Used for time-critical applications
- Used in organizations that employ disciplined methods

Agile Web references

DePaul web site has links to many Agile references

<http://se.cs.depaul.edu/ise/agile.htm>

Extreme Programming - XP

- A very influential agile method, developed in the late 1990s, that introduced a range of agile development techniques by Kent Beck.
- Extreme Programming (XP) takes an ‘extreme’ approach to iterative development.
 - New versions may be built several times per day;
 - Increments are delivered to customers every 2 weeks;
 - All tests must be run for every build and the build is only accepted if tests run successfully.

Extreme Programming - XP

For small-to-medium-sized teams
developing software with vague or rapidly
changing requirements

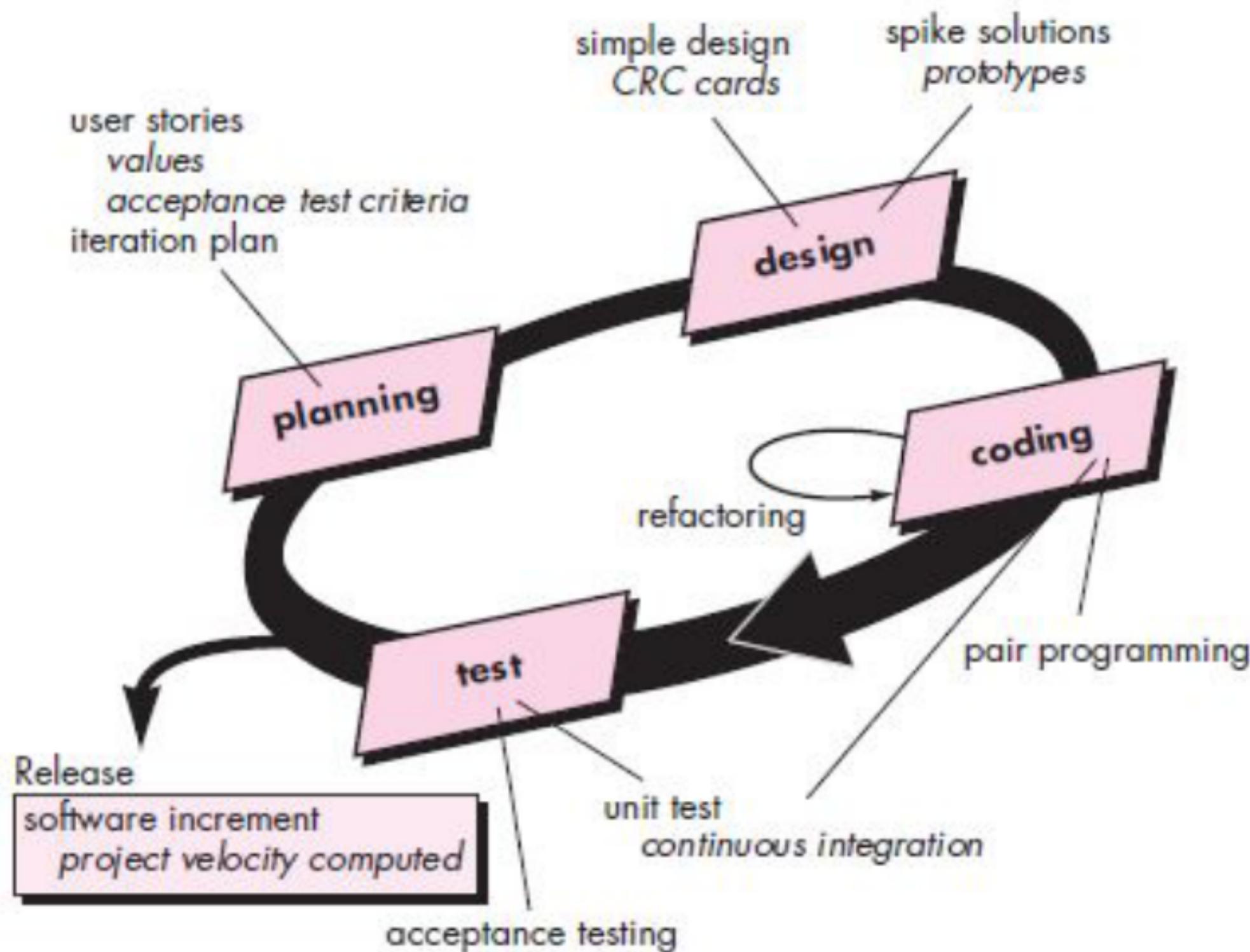
Coding is the key activity throughout a
software project

- Communication among teammates is done with code
- Life cycle and behavior of complex objects defined in test cases – again in code

XP 's 5 values

- Communication (**metaphor/story**),
- Simplicity (**refactoring**),
- Feedback (**software, customer, team members**),
- Courage (**discipline**), and
- respect

The extreme programming release cycle



Extreme programming practices

(a)

Principle or practice	Description
Incremental planning	Requirements are recorded on story cards and the stories to be included in a release are determined by the time available and their relative priority. The developers break these stories into development 'Tasks'. See Figures 3.5 and 3.6.
Small releases	The minimal useful set of functionality that provides business value is developed first. Releases of the system are frequent and incrementally add functionality to the first release.
Simple design	Enough design is carried out to meet the current requirements and no more – KIS principle, Class-Responsibility-Collaborator cards (CRC)
Test-first development	An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented. (Acceptance tests/customer tests)
Refactoring	All developers are expected to refactor the code continuously as soon as possible code improvements are found. This keeps the code simple and maintainable.

Extreme programming practices (b)

Pair programming	Developers work in pairs, checking each other's work and providing the support to always do a good job.
Collective ownership	The pairs of developers work on all areas of the system, so that no islands of expertise develop and all the developers take responsibility for all of the code. Anyone can change anything.
Continuous integration	As soon as the work on a task is complete, it is integrated into the whole system. After any such integration, all the unit tests in the system must pass.
Sustainable pace	Large amounts of overtime are not considered acceptable as the net effect is often to reduce code quality and medium term productivity
On-site customer	A representative of the end-user of the system (the customer) should be available full time for the use of the XP team. In an extreme programming process, the customer is a member of the development team and is responsible for bringing system requirements to the team for implementation.

XP is “extreme” because

Commonsense practices taken to extreme levels

- If code reviews are good, review code all the time (pair programming)
- If testing is good, everybody will test all the time
- If simplicity is good, keep the system in the simplest design that supports its current functionality. (simplest thing that works)
- If design is good, everybody will design daily (refactoring)
- If architecture is important, everybody will work at defining and refining the architecture (metaphor)
- If integration testing is important, build and integrate test several times a day (continuous integration)
- If short iterations are good, make iterations really, really short (hours rather than weeks)

XP and agile principles

- Incremental development is supported through small, frequent system releases.
- Customer involvement means full-time customer engagement with the team.
- People not process through pair programming, collective ownership and a process that avoids long working hours.
- Change supported through regular system releases.
- Maintaining simplicity through constant refactoring of code.

Influential XP practices

- Extreme programming has a technical focus and is not easy to integrate with management practice in most organizations.
- Consequently, while agile development uses practices from XP, the method as originally defined is not widely used.
- Key practices
 - User stories for specification
 - Refactoring
 - Test-first development
 - Pair programming

User stories for requirements

- In XP, a customer or user is part of the XP team and is responsible for making decisions on requirements.
- User requirements are expressed as user stories or scenarios.
- These are written on cards and the development team break them down into implementation tasks. These tasks are the basis of schedule and cost estimates.
- The customer chooses the stories for inclusion in the next release based on their priorities and the schedule estimates.

A ‘prescribing medication’ story

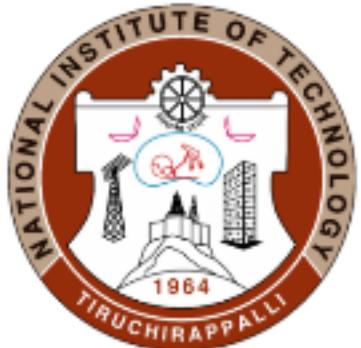
Examples of task cards for prescribing medication

Pair programming

- Pair programming involves programmers working in pairs, developing code together.
- This helps develop common ownership of code and spreads knowledge across the team.
- It serves as an informal review process as each line of code is looked at by more than 1 person.
- It encourages refactoring as the whole team can benefit from improving the system code.

Pair programming

- In pair programming, programmers sit together at the same computer to develop the software.
- Pairs are created dynamically so that all team members work with each other during the development process.
- The sharing of knowledge that happens during pair programming is very important as it reduces the overall risks to a project when team members leave.
- Pair programming is not necessarily inefficient and there is some evidence that suggests that a pair working together is more efficient than 2 programmers working separately.



Software Engineering (CSPE41)

Summer 2023

Lecture 8:

Requirements Engineering

Slides based on various web sources including Pressman's text

Instructor: C. Oswald

Topics covered

- Functional and non-functional requirements
- The software requirements document
- Requirements specification
- Requirements engineering processes
- Requirements elicitation and analysis
- Requirements validation
- Requirements management

Requirements Engineering

- The **process** of establishing the services that the customer requires from a system and the constraints under which it operates and is developed.
- The requirements themselves are the **descriptions** of the system services and **constraints** that are generated during the requirements engineering process.

What is a Requirement?

- It may range from a **high-level abstract statement** of a service or of a system constraint to a **detailed mathematical functional specification**.
- This is inevitable as requirements may serve a dual function
 - May be the basis for a bid for a contract - therefore must be open to interpretation;
 - May be the basis for the contract itself - therefore must be defined in detail;
 - Both these statements may be called requirements.

Requirements Abstraction (Davis)

Types of Requirements

- User requirements

- Statements in natural language plus diagrams of the services the system provides and its operational constraints.
- Written for customers.

- System requirements

- A structured document setting out detailed descriptions of the system's functions, services and operational constraints.
- Defines what should be implemented so may be part of a contract between client and contractor.
- Whom do you think these are written for?
- These are higher level than functional and non-functional requirements, which these may subsume.

The Problems with our Requirements Practices

- We have trouble understanding the requirements that we do acquire from the customer
- We often record requirements in a disorganized manner
- We spend far too little time verifying what we do record
- We **allow change to control us**, rather than establishing mechanisms to control change
- Most importantly, we fail to establish a solid foundation for the system or software that the user wants built (more on next slide)

The Problems with our Requirements Practices (continued)

- Many software developers argue that
 - Building software is so compelling that we want to jump right in (before having a clear understanding of what is needed)
 - Things will become clear as we build the software
 - Project stakeholders will be able to better understand what they need only after examining early iterations of the software
 - Things change so rapidly that requirements engineering is a waste of time
 - The bottom line is producing a working program and that all else is secondary
- All of these arguments contain some truth, especially for small projects that take less than one month to complete
- However, as software grows in size and complexity, these arguments begin to break down and can lead to a failed software project

A Solution: Requirements Engineering (RE)

- Begins during the communication activity and continues into the modeling activity
- Builds a bridge from the system requirements into software design and construction
- Allows the requirements engineer to examine
 - the context of the software work to be performed
 - the specific needs that design and construction must address
 - the priorities that guide the order in which work is to be completed
 - the information, function, and behavior that will have a profound impact on the resultant design

The need for RE

- **Ralph Young** on effective requirements practices, wrote:
 - A customer walks into your office, sits down, looks you straight in the eye, and says,
“I know you think you understand what I said, but what you don’t understand is what I said is not what I meant.”

Example: Recommendation System as a product

RE leads to an understanding of

- what the business impact of the software will be,
- what the customer wants, and
- how end users will interact with the software.

Requirements Engineering Tasks

- Seven distinct tasks
 - Inception
 - Elicitation
 - Elaboration
 - Negotiation
 - Specification
 - Validation
 - Requirements Management
- Some of these tasks may occur in parallel and all are adapted to the needs of the project
- All strive to define what the customer wants
- All serve to establish a solid foundation for the design and construction of the software

Example Project: Campus Information Access Kiosk

- Both podium-high and desk-high terminals located throughout the campus in all classroom buildings, admin buildings, labs, and dormitories
- Hand/Palm-login and logout (seamlessly)
- Voice input
- Optional audio/visual or just visual output
- Immediate access to all campus information plus
 - E-mail
 - Cell phone voice messaging

Inception

Elicitation

Elaboration

Negotiation

Specification

Validation

Requirements
Management

Inception Task

- During inception, the requirements engineer asks a set of questions to establish...
 - A basic **understanding** of the problem
 - The **people** who want a solution
 - The **nature of the solution** that is desired
 - The effectiveness of preliminary **communication** and collaboration between the customer and the developer
- Through these questions, the requirements engineer needs to...
 - Identify the **stakeholders** (Sommerville and Sawyer)
 - Recognize multiple **viewpoints**
 - Work toward **collaboration** (Priority Points)
 - Break the ice and initiate the **communication**

The First Set of Questions

These questions focus on the customer, other stakeholders, the overall goals, and the benefits

- Who is behind the request for this work?
- Who will use the solution?
- What will be the economic benefit of a successful solution?
- Is there another source for the solution that you need?

The Next Set of Questions

These questions enable the requirements engineer to gain a better understanding of the problem and allow the customer to voice his or her perceptions about a solution

- How would you characterize "good" output that would be generated by a successful solution?
- What problem(s) will this solution address?
- Can you show me (or describe) the business environment in which the solution will be used?
- Will special performance issues or constraints affect the way the solution is approached?

The Final Set of Questions

These questions focus on the effectiveness of the communication activity itself

Gause and Weinberg – “meta-questions”

- Are you the **right person** to answer these questions?
Are your answers "official"?
- Are my questions **relevant** to the problem that you have?
- Am I asking **too many** questions?
- Can anyone else provide additional information?
- Should I be asking you anything else?

Inception

Elicitation

Elaboration

Negotiation

Specification

Validation

Requirements
Management

Elicitation Task

- Eliciting requirements is difficult because of
 - Problems of scope in identifying the boundaries of the system or specifying too much technical detail rather than overall system objectives
 - Problems of understanding what is wanted, what the problem domain is, and what the computing environment can handle (Information that is believed to be "obvious" is often omitted)
 - Problems of volatility because the requirements change over time
- Elicitation may be accomplished through two activities
 - Collaborative requirements gathering
 - Quality function deployment

Basic Guidelines of Collaborative Requirements Gathering - FAST

- **Meetings** are conducted and attended by both software engineers, customers, and other interested stakeholders
- **Rules** for preparation and participation are established
- An **agenda** is suggested that is formal enough to cover all important points but informal enough to encourage the free flow of ideas
- A "**facilitator**" (customer, developer, or outsider) controls the meeting
- A "**definition mechanism**" is used such as work sheets, flip charts, wall stickers, electronic bulletin board, chat room, or some other virtual forum
- The goal is to identify the problem, propose elements of the solution, and to discuss different approaches.

Case Study: SafeHome project

- home security function

What will it do?

- The home security function would protect against and/or recognize a variety of undesirable “situations” such as illegal entry, fire, flooding, carbon monoxide levels, and others.
- It’ll use our wireless sensors to detect each situation. It can be programmed by the homeowner, and will automatically telephone a monitoring agency when a situation is detected.
- Objects?
- Services?

mini-specifications

- For *SafeHome* object Control Panel:

The control panel is a wall-mounted unit that is approximately 9 x 5 inches in size. The control panel has wireless connectivity to sensors and a PC. User interaction occurs through a keypad containing 12 keys. A 3 X 3 inch LCD color display provides user feedback. Software provides interactive prompts, echo, and similar functions.

Quality Function Deployment (QFD)

- This is a technique that translates the needs of the customer into technical requirements for software
- It emphasizes an understanding of what is valuable to the customer and then deploys these values throughout the engineering process through functions, information, and tasks
- It identifies three types of requirements
 - Normal requirements: These requirements are the objectives and goals stated for a product or system during meetings with the customer (e.g. **graphical displays, specific system functions, and defined levels of performance**)
 - Expected requirements: These requirements are implicit to the product or system and may be so fundamental that the customer does not explicitly state them (e.g., **HCI**)

Elicitation Work Products

The work products will vary depending on the system, but should include one or more of the following items

- A statement of need and feasibility
- A bounded statement of scope for the system or product
- A list of customers, users, and other stakeholders who participated in requirements elicitation
- A description of the system's technical environment
- A list of requirements (organized by function) and the domain constraints that apply to each
- A set of preliminary usage scenarios (in the form of use cases) that provide insight into the use of the system or product under different operating conditions

Inception

Elicitation

Elaboration

Negotiation

Specification

Validation

Requirements
Management

Elaboration Task

- During elaboration, the software engineer takes the information obtained during inception and elicitation and begins to expand and refine it
- Elaboration focuses on developing a refined technical model of software functions, features, and constraints
- It is an analysis modeling task
 - **Use cases** are developed
 - **Domain classes** are identified along with their attributes and relationships
 - **State machine diagrams** are used to capture the life on an object
- The end result is an analysis model that defines the functional, informational, and behavioral domains of the problem

Developing Use Cases

- Step One – Define the **set of actors** that will be involved in the story
 - Actors **are people, devices, or other systems** that use the system or product within the context of the function and behavior that is to be described
 - Actors are anything that **communicate with the system** or product and that are external to the system itself
- Step Two – Develop use cases, where each one answers a set of questions

(More on next slide)

Actors and Users

- an actor and an end user are not necessarily the same
- an actor represents a class of external entities (often, but not always, people) that play just one role in the context of the use case
- consider a **machine operator** (a user) who interacts with the control computer for a manufacturing cell.
- Four modes: programming mode, test mode, monitoring mode, and troubleshooting mode
- Four actors: programmer, tester, monitor, and troubleshooter.
- Primary actors - interact to achieve required system function
- Secondary actors - support the system so that primary actors can do their work.

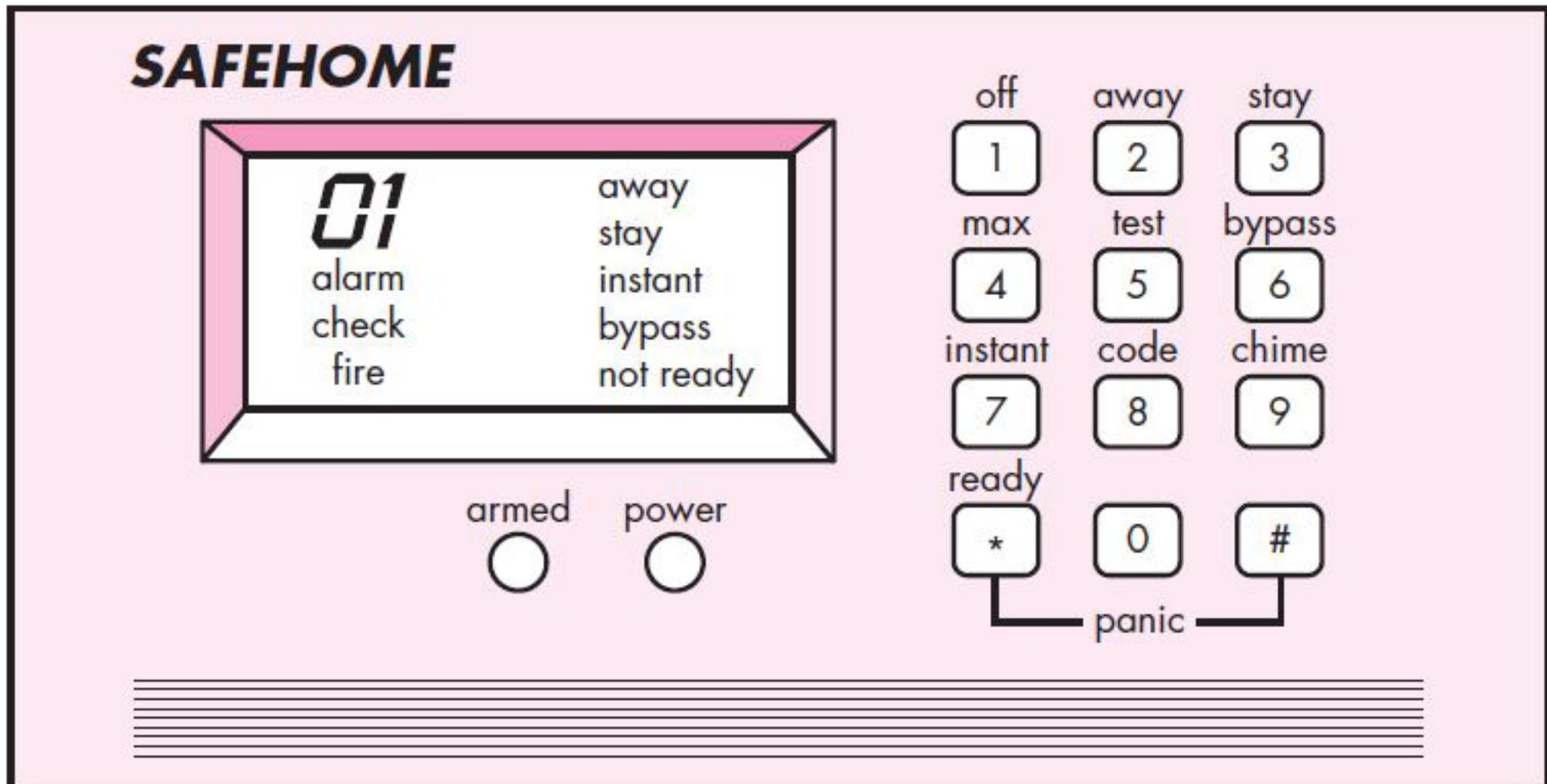
SafeHome actors

- **Homeowner** (a user),
- **setup manager** (likely the same person as **homeowner**, but playing a different role),
- **sensors** (devices attached to the system), and the
- **monitoring and response subsystem** (the central station that monitors the *SafeHome* home security function).

homeowner actor:

- Enters a password to allow all other interactions.
- Inquires about the status of a security zone.
- Inquires about the status of a sensor.
- Presses the panic button in an emergency.
- Activates/deactivates the security system.

SafeHome Control Panel



Use Case Template

Use case: *InitiateMonitoring*

Primary actor: Homeowner.

Goal in context: To set the system to monitor sensors when the

homeowner leaves the house or remains inside.

Preconditions: System has been programmed for a password and to recognize various sensors.

Trigger: The homeowner decides to “set” the system, i.e., to turn on the alarm functions.

Scenario:

1. Homeowner: observes control panel
2. Homeowner: enters password
3. Homeowner: selects “stay” or “away”
4. Homeowner: observes read alarm light to indicate that

Use Case Template contd'

Exceptions:

- Control panel is *not ready*: homeowner checks all sensors to determine which are open; closes them.
- Password is incorrect (control panel beeps once): homeowner reenters correct password.
- Password not recognized: monitoring and response subsystem must be contacted to reprogram password.
- *Stay* is selected: control panel beeps twice and a *stay* light is lit; perimeter sensors are activated.
- *Away* is selected: control panel beeps three times and an *away* light is lit; all sensors are activated.

Use Case Template contd'

- **Priority:** Essential, must be implemented
- **When available:** First increment
- **Frequency of use:** Many times per day
- **Channel to actor:** Via control panel interface
- **Secondary actors:** Support technician, sensors
- **Channels to secondary actors:**
 - Support technician: phone line; Sensors: hardwired and radio frequency interfaces

Open issues:

1. Should there be a way to activate the system without the use of a password or with an abbreviated password?
2. Should the control panel display additional text messages?

UML diagrams

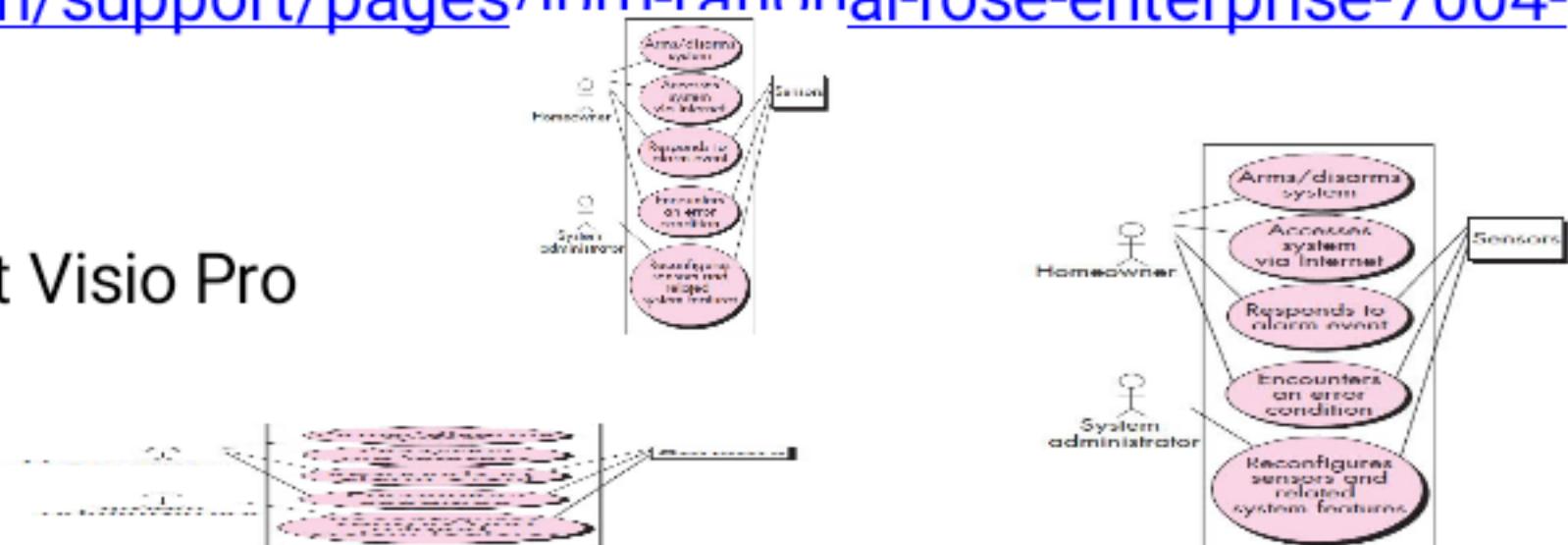
- A UML diagram is a diagram based on the UML (Unified Modeling Language) with the purpose of visually representing a system along with its main actors, roles, actions, artifacts or classes, in order to better understand, alter, maintain, or document information about the system.
- **What is UML?**
 - A general purpose modeling language
 - intended to provide a standard way to visualize the design of a system.

Software:

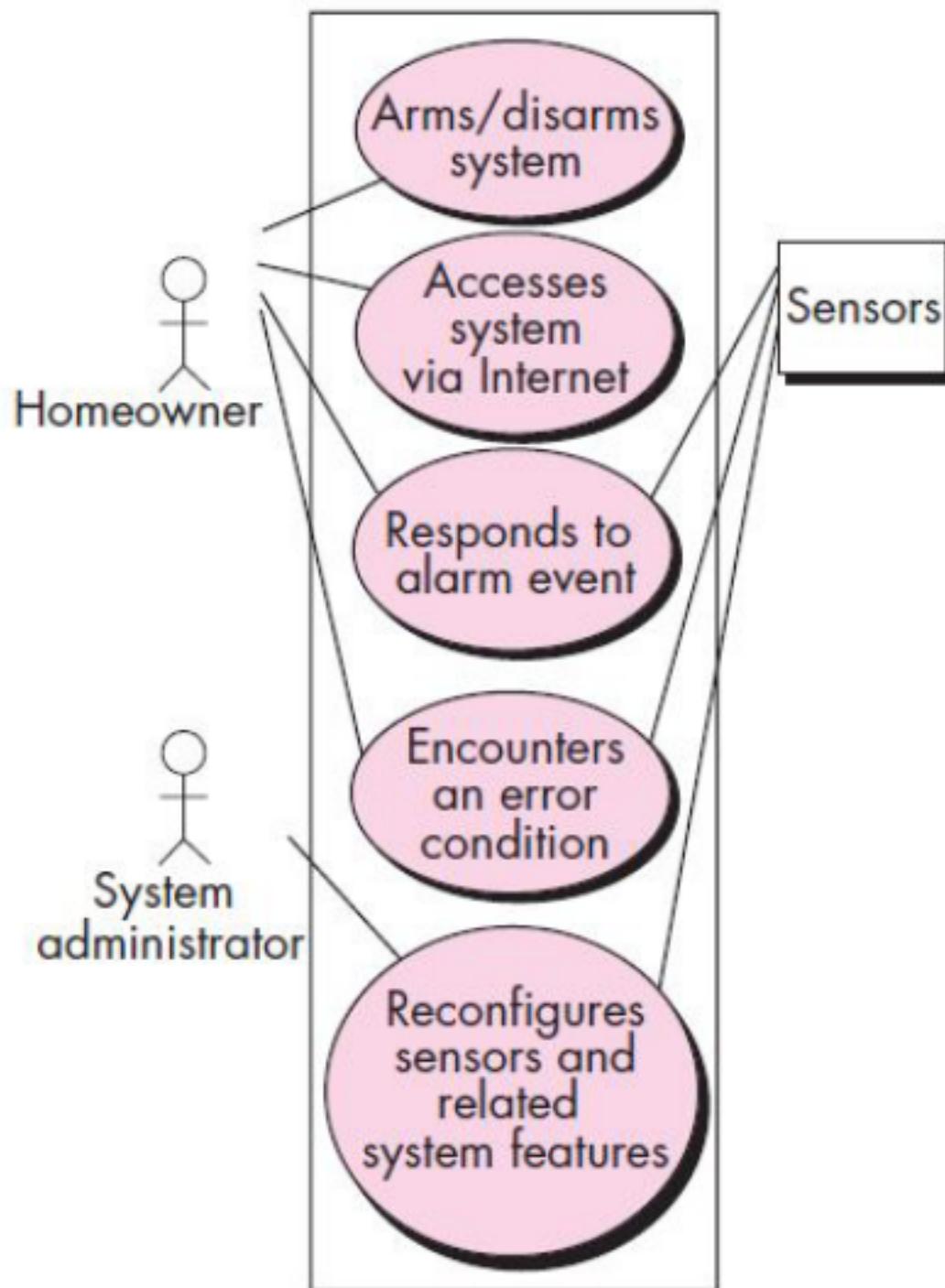
<https://www.ibm.com/support/pages/ihm-rational-rose-enterprise-7004-ifix001>

Others:

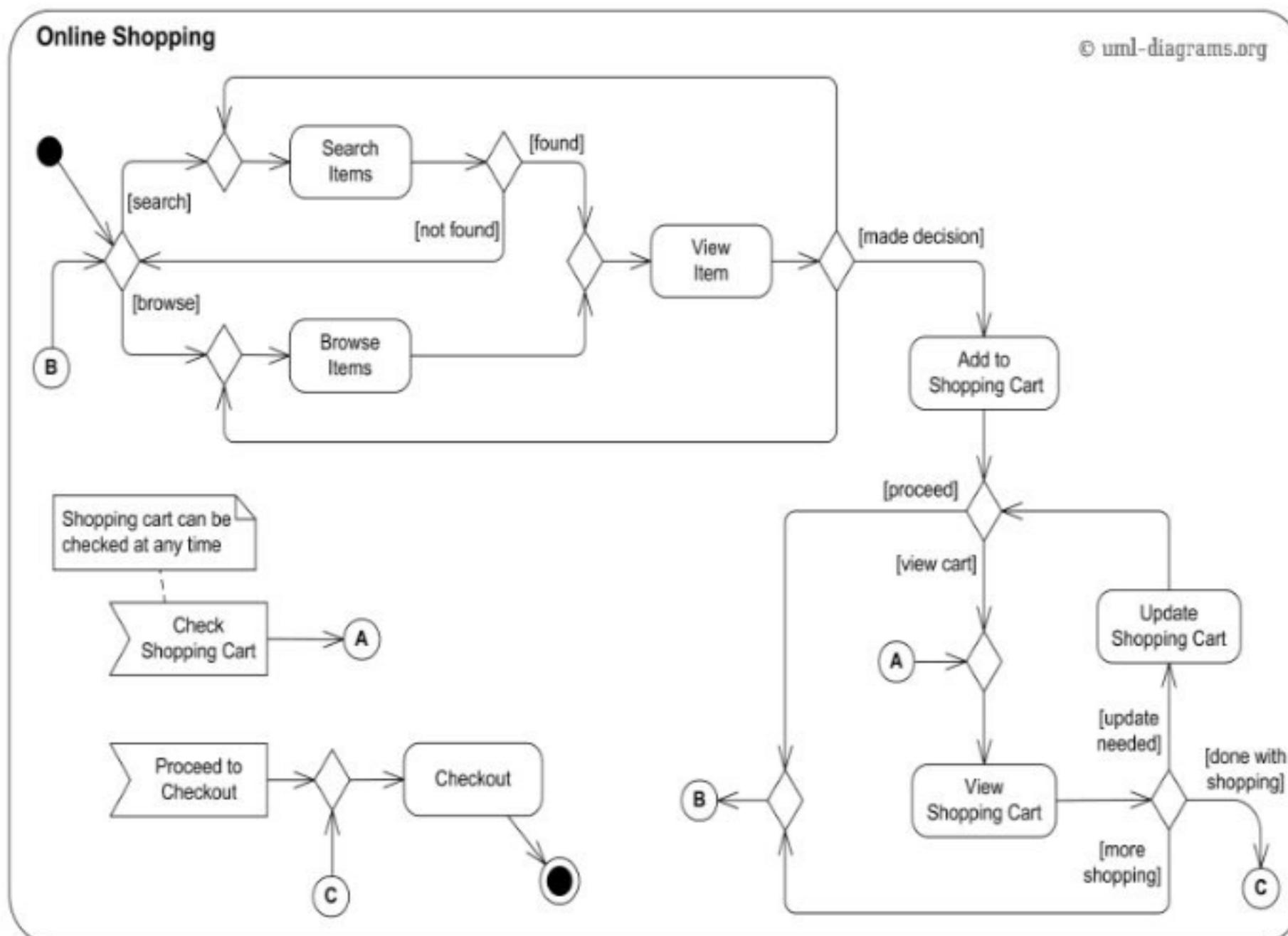
Lucidchart, Microsoft Visio Pro



UML use case diagram for *SafeHome* home security function

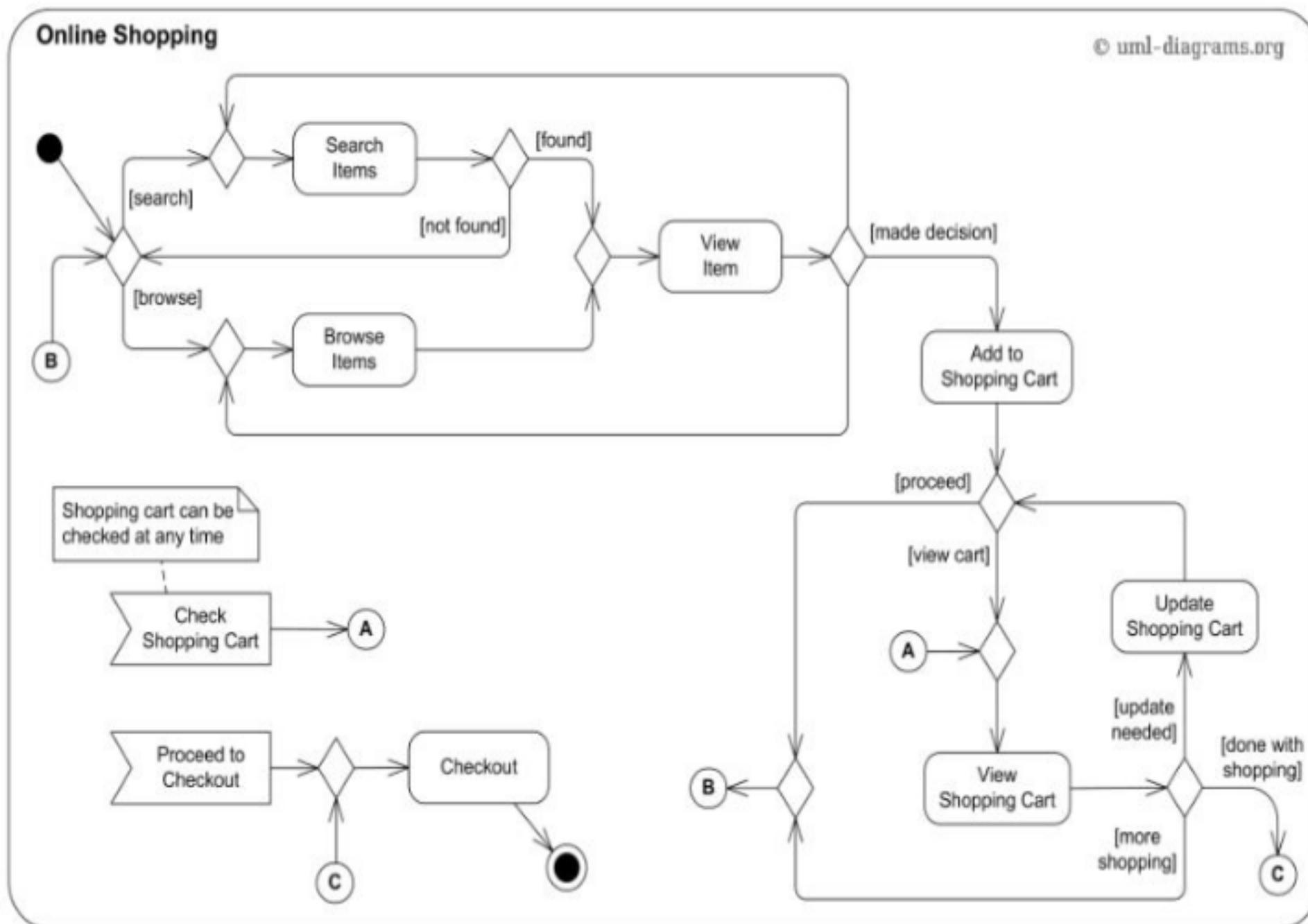


Usecase diagram for Online Shopping Tool

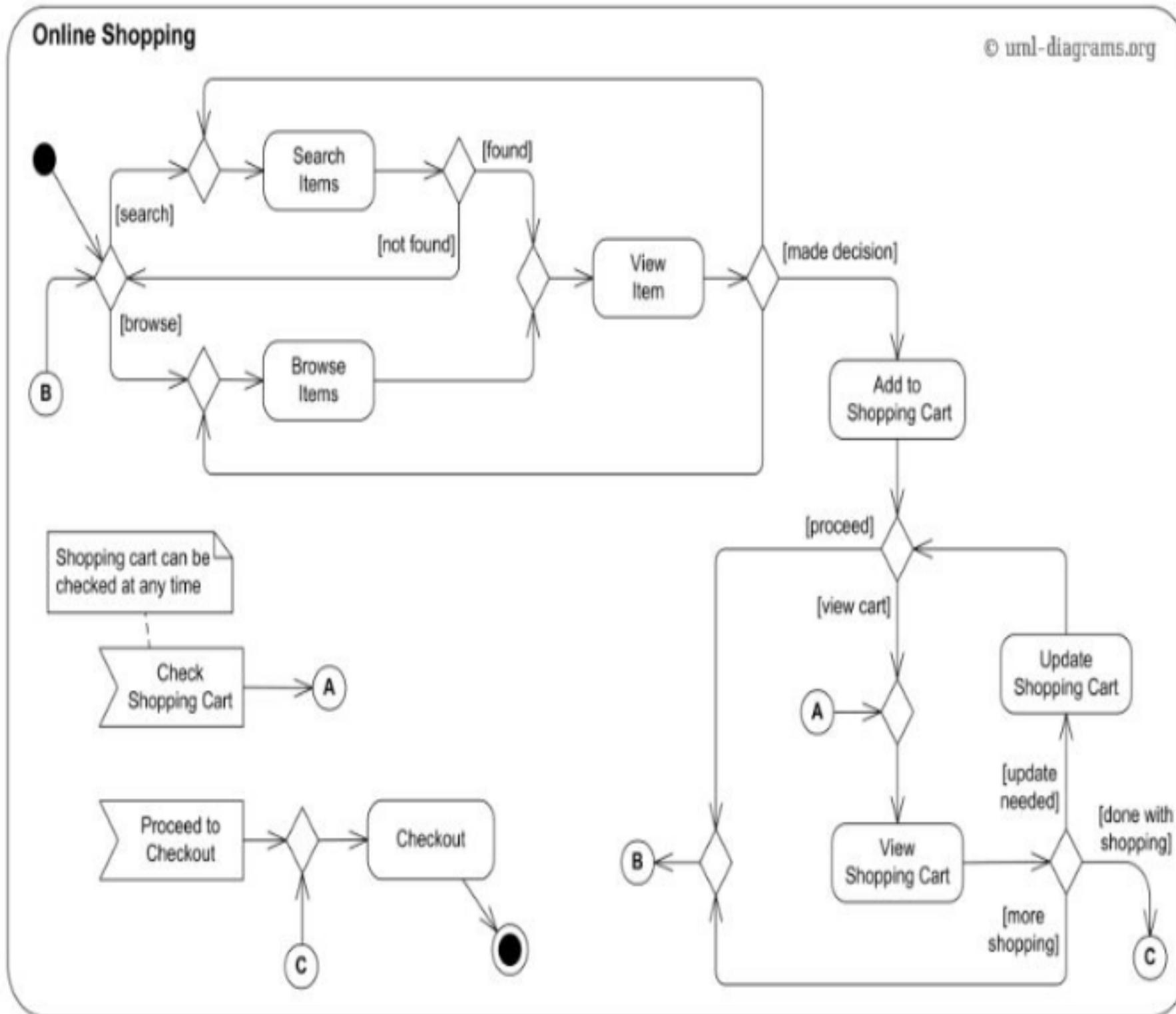


<https://www.uml-diagrams.org/examples/online-shopping-use-case-diagram-example.html>

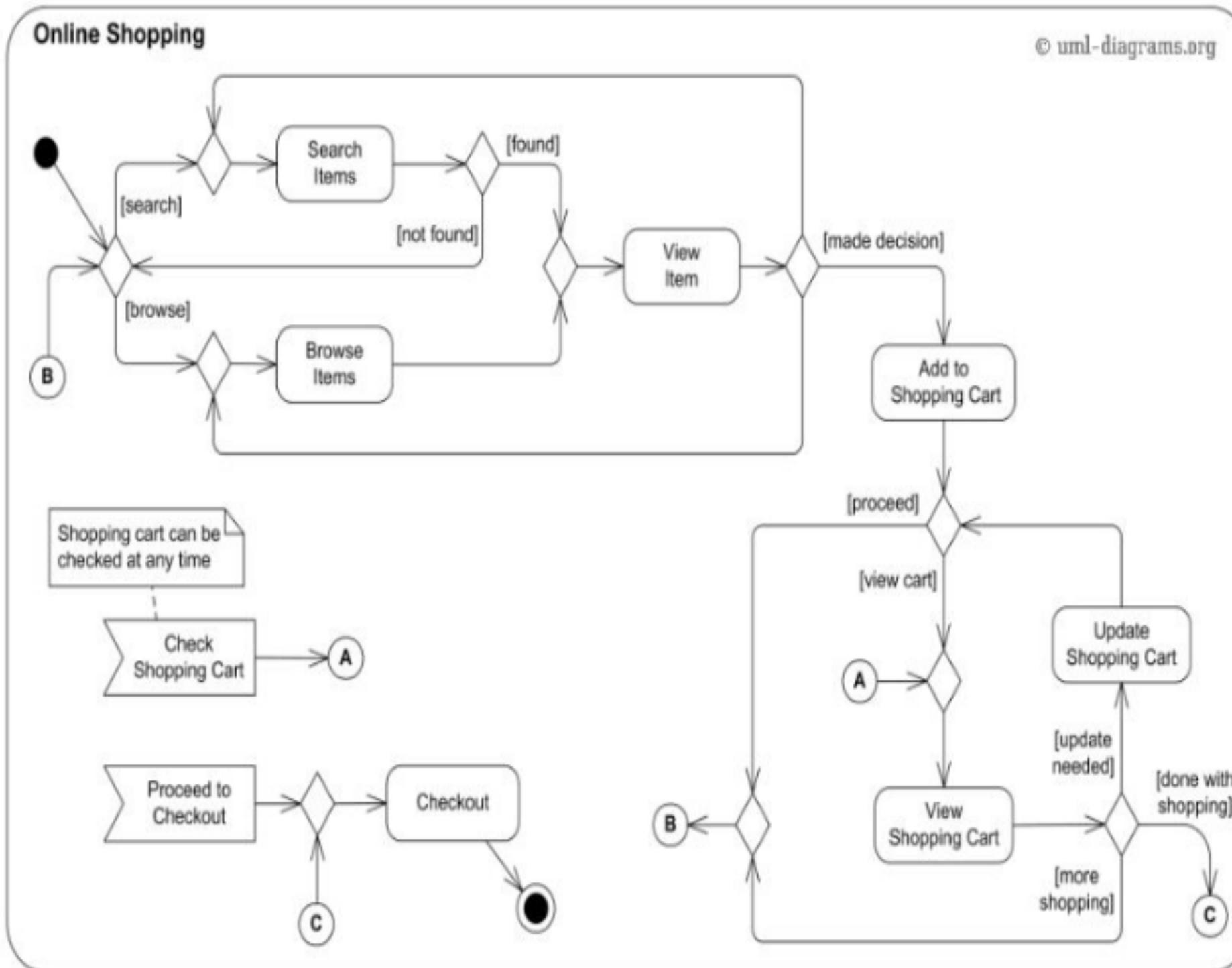
Online shopping use case diagram example - view items use case.



Online shopping UML use case diagram example - checkout, authentication and payment use cases



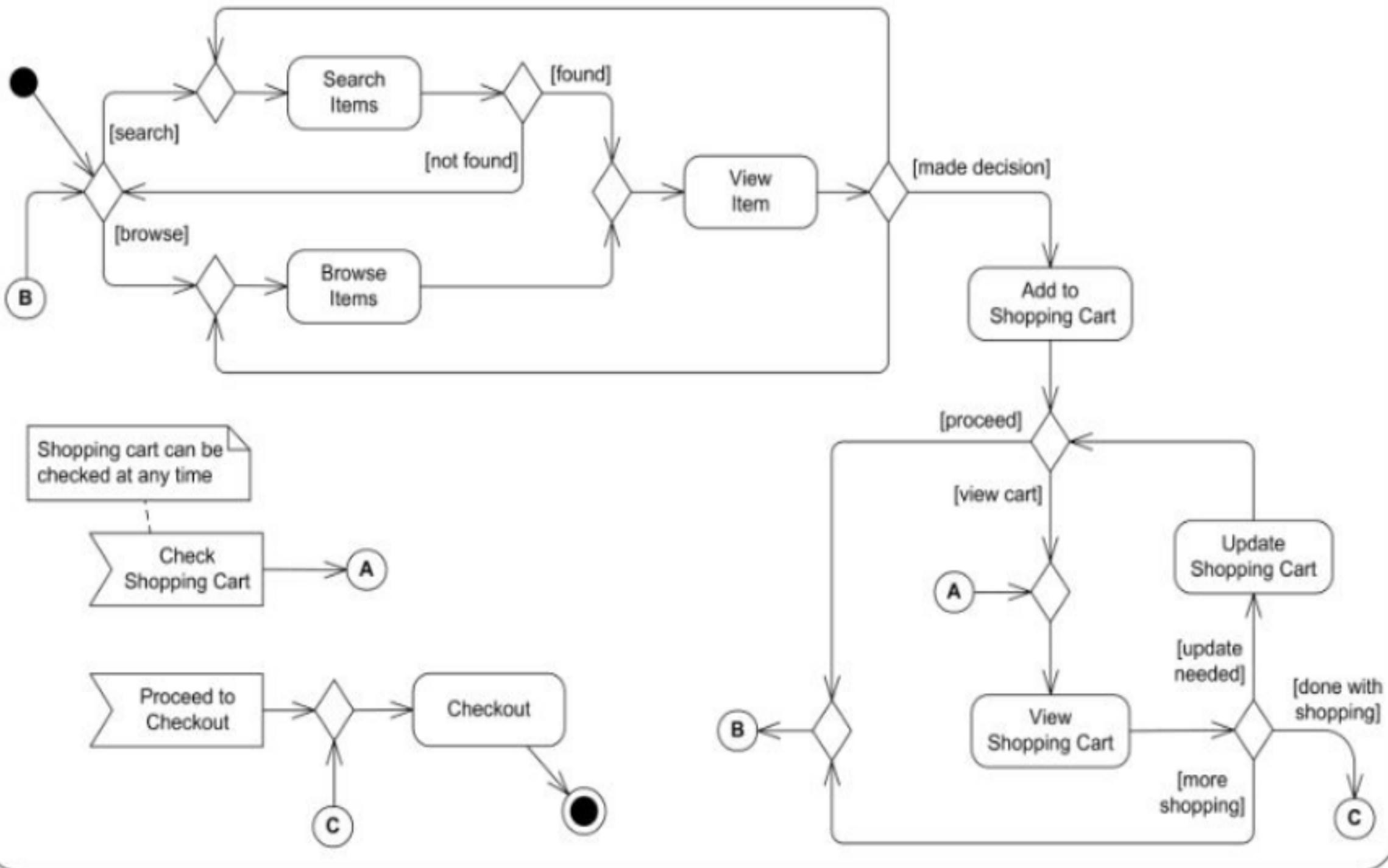
UML activity diagrams for eliciting requirements



UML activity diagram for online shopping

Online Shopping

© uml-diagrams.org



Questions Commonly Answered by a Use Case - Jacobson

- Who is the primary actor(s), the secondary actor(s)?
- What are the actor's goals?
- What preconditions should exist before the scenario begins?
- What main tasks or functions are performed by the actor?
- What exceptions might be considered as the scenario is described?
- What variations in the actor's interaction are possible?
- What system information will the actor acquire, produce, or change?
- Will the actor have to inform the system about changes in the external environment?
- What information does the actor desire from the system?
- Does the actor wish to be informed about unexpected

Workout

Develop usecase and activity diagrams for :

Oddnos: Pragyan web portal development

Even nos: Plagiarism checking system

Draw it in your note book first and realize it in the tool

Elements of the Analysis Model

- Scenario-based elements
 - Describe the system from the user's point of view using scenarios that are depicted in use cases and activity diagrams
- Class-based elements
 - Identify the domain classes for the objects manipulated by the actors, the attributes of these classes, and how they interact with one another; they utilize class diagrams to do this
- Behavioral elements
 - Use state diagrams to represent the state of the system, the events that cause the system to change state, and the actions that are taken as a result of a particular event; can also be applied to each class in the system
- Flow-oriented elements
 - Use data flow diagrams to show the input data that

Inception

Elicitation

Elaboration

Negotiation

Specification

Validation

Requirements
Management

Negotiation Task

- During negotiation, the software engineer reconciles the conflicts between what the customer wants and what can be achieved given limited business resources
- Requirements are **ranked** (i.e., prioritized) by the customers, users, and other stakeholders
- **Risks** associated with each requirement are identified and analyzed
- Rough guesses of development effort are made and used to assess the impact of each requirement on project cost and delivery time
- Using an **iterative** approach, requirements are eliminated, combined and/or modified so that each party achieves some measure of satisfaction

The Art of Negotiation

- Recognize that it is not competition
- Map out a strategy
- Listen actively
- Focus on the other party's interests
- Don't let it get personal
- Be creative
- Be ready to commit

Inception

Elicitation

Elaboration

Negotiation

Specification

Validation

Requirements
Management

Specification Task

- A specification is the final work product produced by the requirements engineer
- It is normally in the form of a **software requirements specification (SRS)**
- It serves as the foundation for subsequent software engineering activities
- It describes the function and performance of a computer-based system and the constraints that will govern its development
- It formalizes the informational, functional, and behavioral requirements of the proposed software in both a graphical and textual format

Design Engineering

Slides adapted from various web sources with grateful acknowledgement of the many others who made their course materials freely available online.

Five Notable Design Quotes

- "Questions about whether design is necessary or affordable are quite beside the point; design is inevitable. The alternative to good design is bad design, [rather than] no design at all." **Douglas Martin**
- "You can use an eraser on the drafting table or a sledge hammer on the construction site." **Frank Lloyd Wright**
- "The public is more familiar with bad design than good design. If is, in effect, conditioned to prefer bad design, because that is what it lives with; the new [design] becomes threatening, the old reassuring." **Paul Rand**
- "A common mistake that people make when trying to design something completely foolproof was to underestimate the ingenuity of complete fools." **Douglas Adams**
- "Every now and then go away, have a little relaxation, for when you come back to your work your judgment will be surer. Go some distance away because then the work appears smaller and more of it can be taken in at a glance and a lack of harmony and proportion is more readily seen." **Leonardo DaVinci**

Introduction

The Process of Design

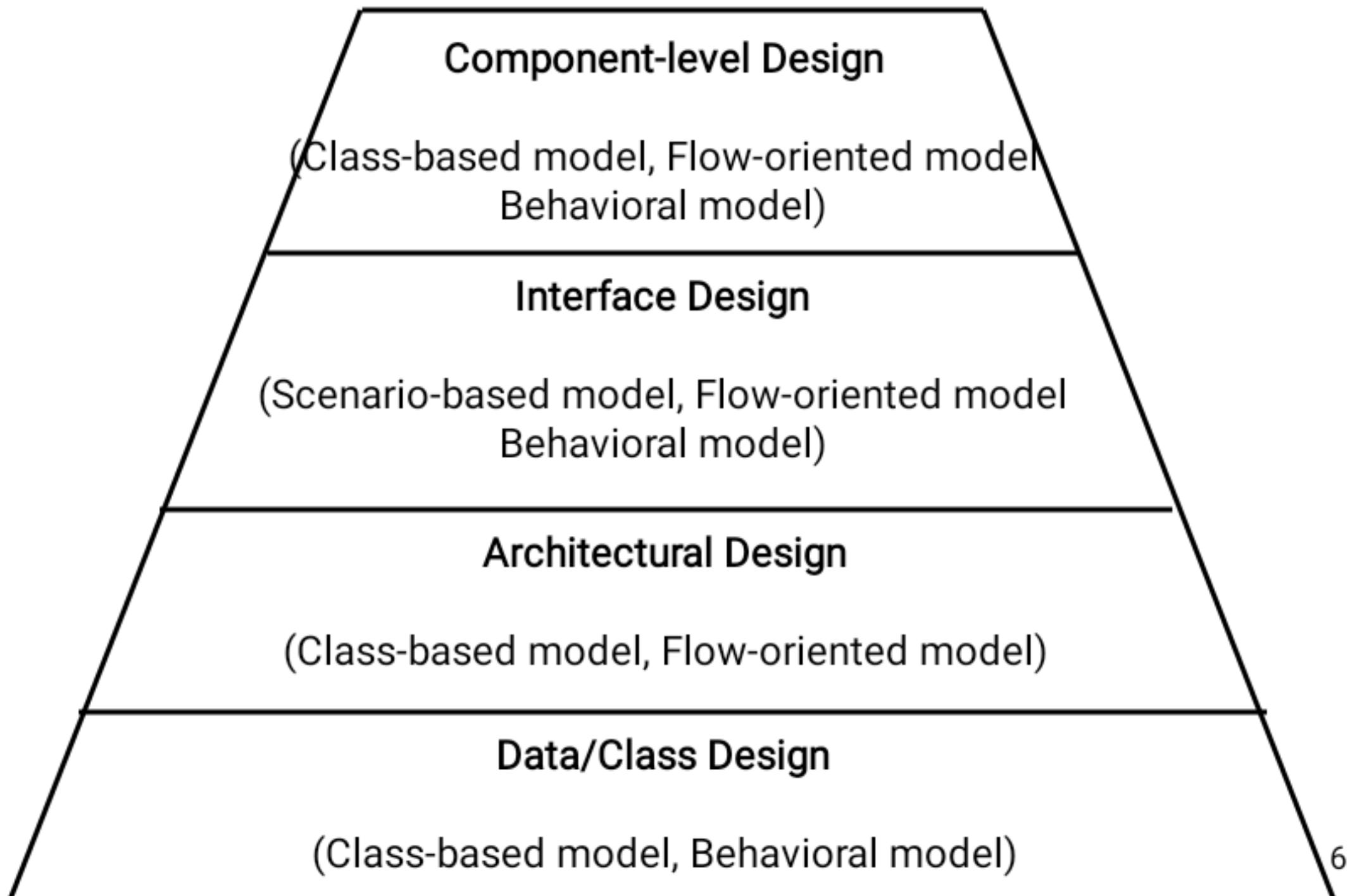
- Definition:

- *Design* is a problem-solving process whose objective is to find and describe a way:
 - To implement the system's *functional requirements*...
 - ...while respecting the constraints imposed by the *non-functional requirements*...
 - Such as performance, maintainability, security, persistence, cost, reliability, portability, etc.....(long list)
 - including also the budget, technologies, environment, legal issues, deadlines, ...
 - And while adhering to general principles of *good quality*

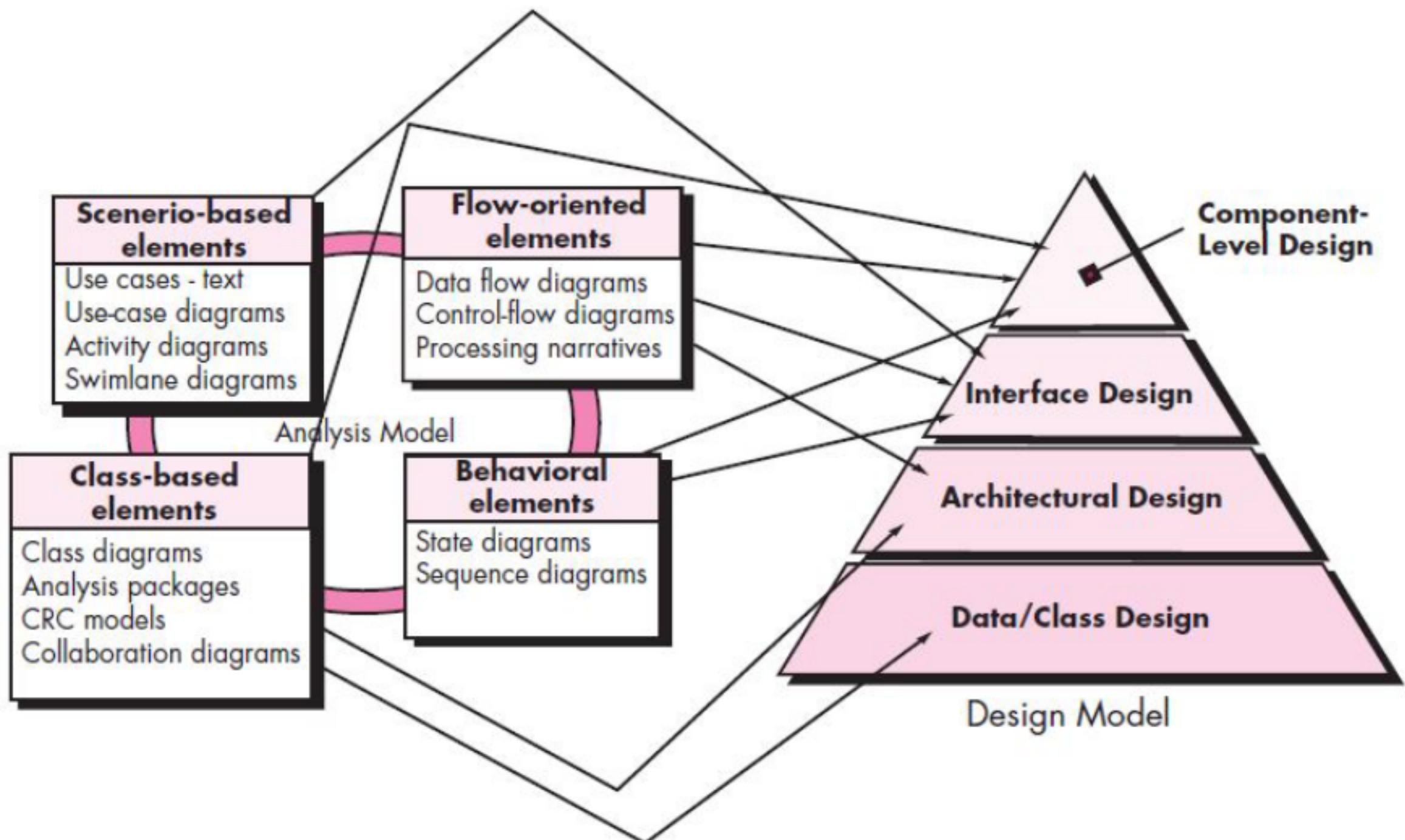
Why design?

- In 1990s **Mitch Kapor**, the creator of Lotus 1-2-3, presented a “software design manifesto” said:
 - “**It’s where you stand with a foot in two worlds—the world of technology and the world of people and human purposes—and you try to bring the two together. . .**”
- Roman architecture critic –
 - *Firmness*: A program should **not have any bugs** that inhibit its function.
 - *Commodity*: A program should be **suitable for the purposes** for which it was intended.
 - *Delight*: The experience of using the program should be a **pleasurable** one.
- Software design is the last software engineering action within the modeling activity and sets the stage for **construction** (code generation and testing).
- Software design can be stated with a single word—***quality***.

From Analysis Model to Design Model (continued)



Translating the requirements model into the design model



The Design Process

- Software design is an iterative process through which requirements are translated into a “blueprint” for constructing the software.

Software Quality Guidelines and Attributes

McGlaughlin suggests:

- design must implement all of the **explicit requirements**
- design must be a **readable, understandable** guide for those who generate Code and test
- should provide a complete picture of the software, addressing the data, functional, and behavioral domains

Quality Guidelines

- Modular
- architectural styles or patterns
- distinct representations of data, architecture, interfaces, and components.
- should lead to data structures that are appropriate
- and few more..

Hewlett-Packard's view (FURPS):

- **Functionality** (what the system does and the purpose of the system)
- **Usability** (considering human factors)
- **Reliability** (frequency and severity of failure)
- **Performance** (space and time)
- **Supportability** (maintainability)

The Evolution of Software Design

- Procedural aspects of design definition evolved into a philosophy - *structured programming*

M. A. Jackson: “The beginning of wisdom for a [software engineer] is to recognize the difference between getting a program to work, and getting it right.”

Design Concepts:

Abstraction

- **Procedural abstraction** – a sequence of instructions that have a specific and limited function Eg. Word ‘open’ for a door. Open -> long sequence of procedural steps
- **Data abstraction** – a named collection of data that describes a data object - Eg. Door – encompass set of attributes that describes a door (door type, swing direction, opening mechanism, weight, dimensions etc.)

Design Concepts

- **Architecture**

- The overall structure of the software and the ways in which the structure provides conceptual integrity for a system
- Consists of components, connectors, and the relationship between them
- *Structural models, Framework models, Dynamic models, Process models and Functional models*

- **Patterns**

- a general **reusable** solution to a commonly occurring problem in software design.

A design pattern isn't a finished design that can be transformed directly into code - a **description or template** for how to solve a problem that can be used in many different situations.

- A design structure that solves a particular design problem within a specific context

Design Concepts (continued)

- It provides a description that enables a designer to determine whether the pattern is applicable, whether the pattern can be reused, and whether the pattern can serve as a guide for developing similar patterns
- “Distilled wisdom” about object-oriented programming

Separation of Concerns

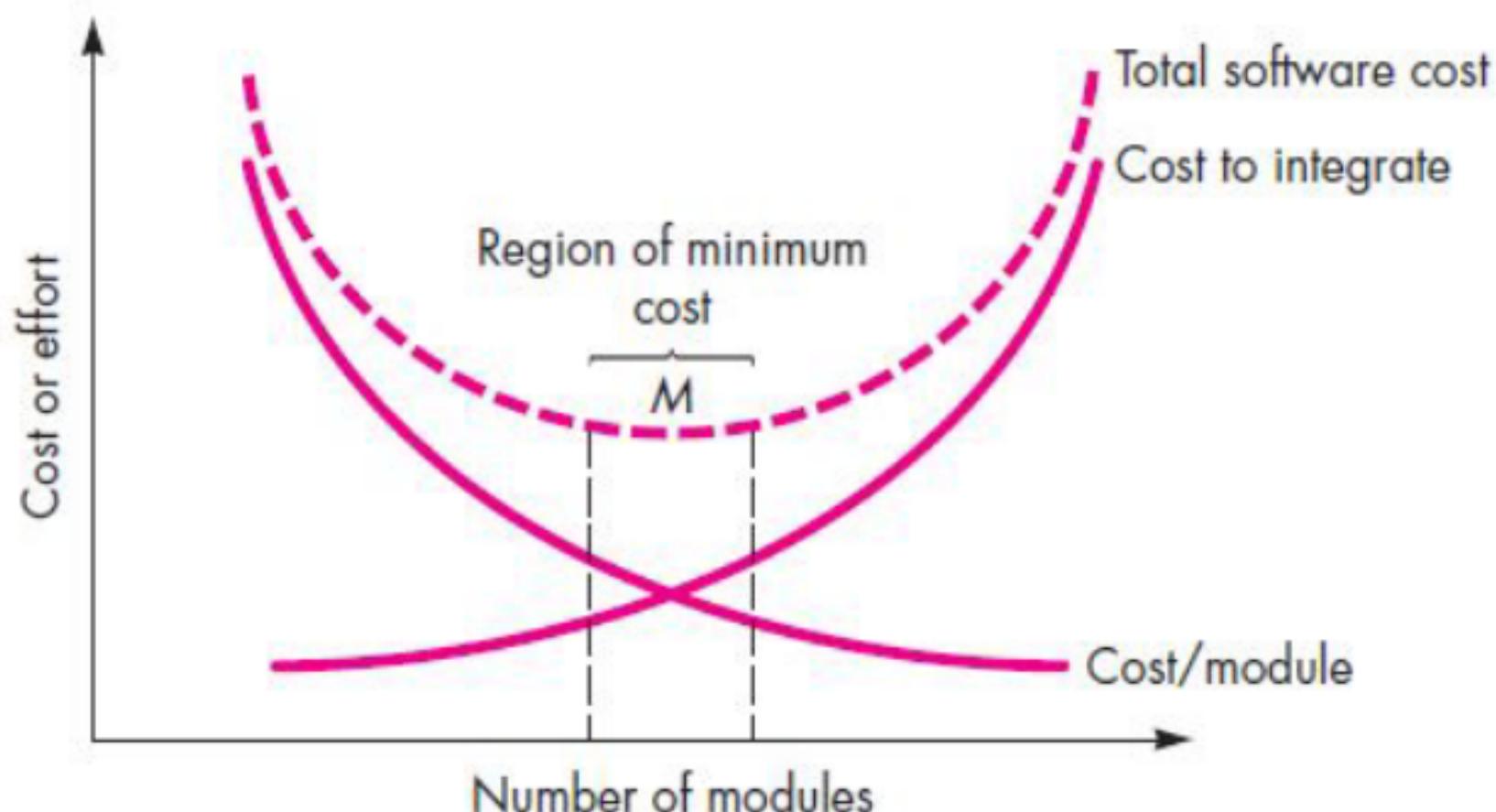
- suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently

concern is a feature or behavior that is specified as part of the requirements model for the software.
(more on next slide)

Design Concepts (continued)

● Modularity

- Separately named and addressable components (i.e., modules) that are integrated to satisfy requirements (divide and conquer principle)
- Makes software intellectually manageable so as to grasp the control paths, span of reference, number of variables, and overall complexity



Design Concepts (continued)

- Information hiding

- The designing of modules so that the algorithms and local data contained within them are inaccessible to other modules
 - This enforces access constraints to both procedural (i.e., implementation) detail and local data structures

- Functional independence (**Wirth and Parnas**)

- Modules that have a "single-minded" function and an aversion to excessive interaction with other modules
 - Independent modules – easier to develop because function can be compartmentalized and interfaces are simplified.
 - High cohesion – a module performs only a single task
 - Low coupling – a module has the lowest amount of connection needed with other modules
(more on next slide)

Abstract Data Types

- Def.
 - data **type** together with actions to be performed on **instantiations** of that data type
 - subtle difference with encapsulation
- Example

```
class JobQueue {  
    public int queueLength; // length of job queue  
    public int queue = new int[25]; // up to 25 jobs  
    // methods  
    public void initializeJobQueue (){  
        // body of method  
    }  
    public void addJobToQueue (int jobNumber){  
        // body of method  
    }  
} // JobQueue
```

Information Hiding

- Really “details hiding”
 - hiding implementation details, not information
- Example
 - design modules so that items likely to change are hidden
 - future change localized
 - change cannot affect other modules
 - data abstraction
 - designer thinks at level of ADT

Information hiding

- Example

```
class JobQueue {  
    private int queueLength; // length of job queue  
    private int queue = new int[25]; // up to 25 jobs  
    // methods  
    public void initializeJobQueue(){  
        // body of method  
    }  
    public void addJobToQueue (int jobNumber){  
        // body of method  
    }  
} // JobQueue
```

- Now **queue** and **queueLength** are inaccessible

Design Concepts (continued)

- **Stepwise refinement** (by **Niklaus Wirth**)

- Development of a program by successively refining levels of procedure detail
 - Complements abstraction, which enables a designer to specify procedure and data and yet suppress low-level details

- **Refactoring** (**Fowler**)

- A reorganization technique that simplifies the design (or internal code structure) of a component without changing its function or external behavior
 - Removes redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failures (<https://refactoring.com/>)

- **Design classes**

- Refines the analysis classes by providing design detail that will enable the classes to be implemented
 - Creates a new set of design classes that implement a software infrastructure to support

Design Concepts (continued)

Aspects

An *aspect* is a representation of a crosscutting concern

- Consider two requirements, *A* and *B*. Requirement *A* *crosscuts requirement B* “if a software decomposition [refinement] has been chosen in which *B* cannot be satisfied without taking *A* into account”

Example: SafeHomeAssured.com WebApp.

- Requirement *A* - enable a registered user to access video from cameras placed throughout a space
- Requirement *B* - *registered user must be validated prior to using the website*
 - *A** is a design representation for requirement *A* and *B** is a design representation for requirement *B*.
 - Therefore, *A** and *B** are representations of concerns, and *B** *crosscuts A** - the design representation, *B** - is an aspect of the app.

Modules

- What is a module?

- lexically contiguous sequence of program statements, bounded by boundary elements, with aggregate identifier

- Examples

- procedures & functions in classical PLs
 - objects & methods within objects in OO PLs

Good vs. Bad Modules

- Modules in themselves are not “good”
- Must design them to have good properties

Cohesion and Coupling (Yourdon and Constantine in 1979)

- **Cohesion** - an indication of the relative functional strength of a module (qualitative indication of the degree to which a module focuses on just one thing.)
- **Coupling** - indication of the relative interdependence among modules (qualitative indication of the degree to which a module is connected to other modules and to the outside world.)

Cohesion:

- The degree to which all elements of a component are directed towards a single task and all elements directed towards that task are contained in a single component (S.L.Pfleeger).
- Cohesive module performs a single task, requiring little interaction with other components in other parts of a program.
- “Schizophrenic” components (modules that perform many different tasks)

Good vs. Bad Modules

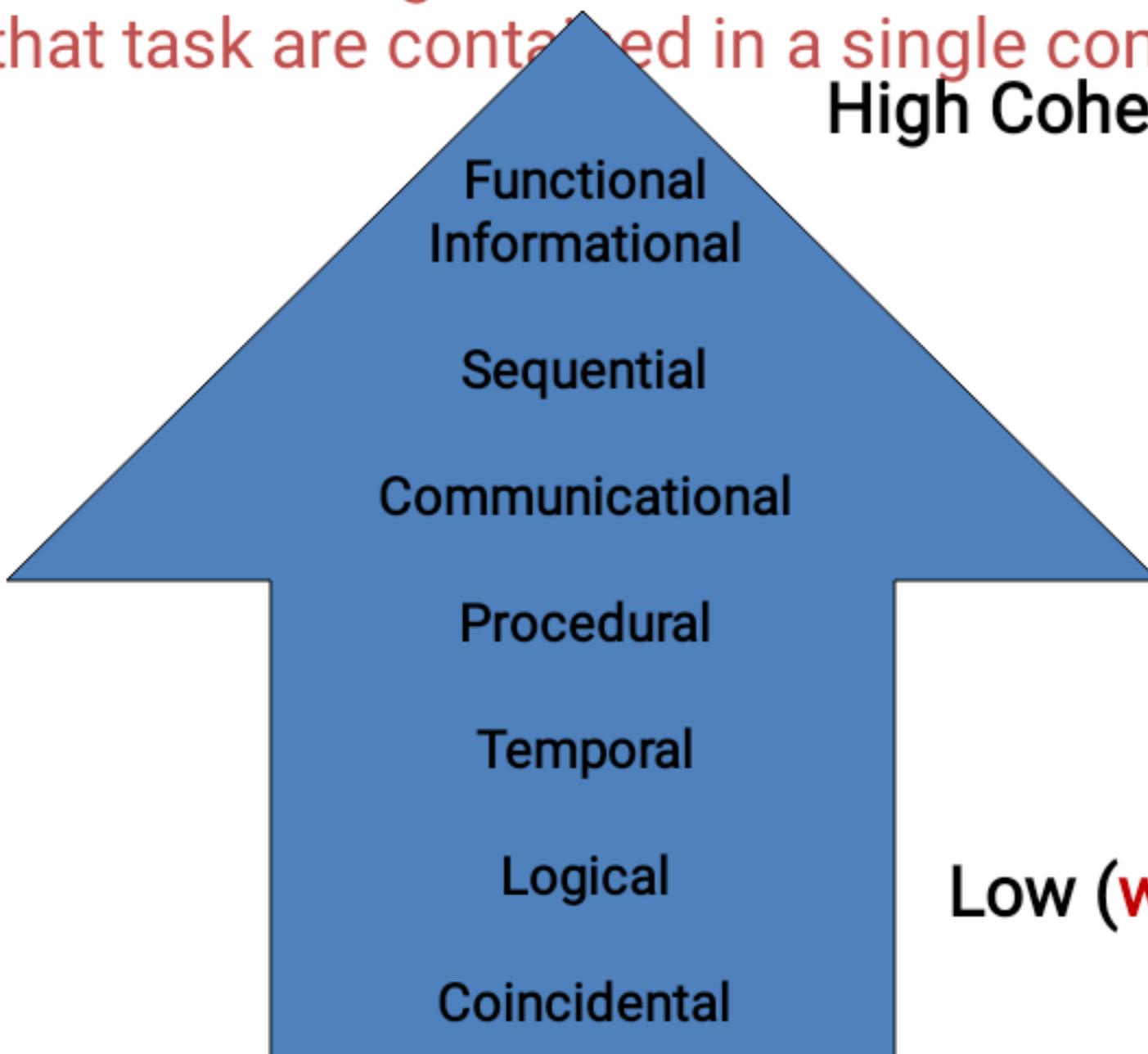
- Two designs functionally equivalent, but the 2nd is
 - hard to understand
 - hard to locate faults
 - difficult to extend or enhance
 - cannot be reused in another product. Implication?
 - -> expensive to perform maintenance
- Good modules must be like the 1st design
 - maximal relationships within modules (cohesion)
 - minimal relationships between modules (coupling)
 - this is the main contribution of structured design
- Exception identification and handling
- Fault prevention and fault tolerance

Range of Cohesion

Degree of interaction **within** module

The degree to which all elements of a component are directed towards a single task and all elements directed towards that task are contained in a **single** component.

High Cohesion (best)



1. Coincidental Cohesion

- Def. ?
 - module performs multiple, completely unrelated actions/
Parts of the component are only related by their location in source code
- Example
 - module prints next line, reverses the characters of the 2nd argument, adds 7 to 3rd argument
- How could this happen?
 - hard organizational rules about module size
- Why is this bad?
 - degrades maintainability & modules are not reusable
- Easy to fix. How?
 - break into separate modules each performing one task

2. Logical Cohesion

- Def.
 - module performs series of related actions, one of which is selected by calling module/**Elements of component are related logically and not functionally.**
- Example
 - A component reads inputs from tape, disk, and network. All the code for these functions are in the same component.
- Why is this bad?
 - Operations are related, but the functions are significantly different.
 - interface difficult to understand
 - code for more than one action may be intertwined
 - difficult to reuse
- How to fix?

A device component has a read operation that is overridden by sub-²⁶ class components. The tape sub-class reads from tape. The disk

3. Temporal Cohesion

- Def. ?
 - module performs series of actions related in time
- Initialization example
 - 1. open old db, new db, transaction db, print db, initialize sales district table, read first transaction record, read first old db record
 - 2. A system initialization routine: this routine contains all of the code for initializing all of the parts of the system. Lots of different activities occur, all at init time.
- Why is this bad?
 - actions weakly related to one another, but strongly related to actions in other modules
 - code spread out -> not maintainable or reusable
- Initialization example fix
 - define these initializers in the proper modules & then have an initialization module call each

4. Procedural Cohesion

- Def. ?
 - module performs series of actions related by procedure to be followed by product
- Example
 - update part number and update repair record in master db
- Why is this bad?
 - actions are still weakly related to one another
 - not reusable
- Solution
 - break up!

5. Communicational Cohesion

- Def.
 - module performs series of actions related by procedure to be followed by product, but in addition all the actions operate on same data
- Example 1
 - update record in db and send it to printer**

```
database.Update (record).  
record.Print().
```

- Example 2
 - calculate new coordinates and send *them* to window**
- Why is this bad?
 - still leads to less reusability -> break it up

6. Sequential Cohesion

- The output of one component is the input to another.
- Occurs naturally in functional programming languages
- Good situation

7. Informational Cohesion

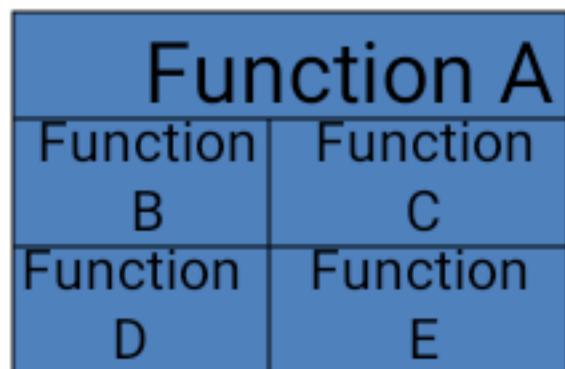
- Def.
 - module performs a number of actions, each with its own entry point, with independent code for each action, all performed on the same data structure
- Different from logical cohesion
 - Each piece of code has single entry and single exit
 - In logical cohesion, actions of module intertwined
- ADT and object-oriented paradigm promote

7. Functional Cohesion

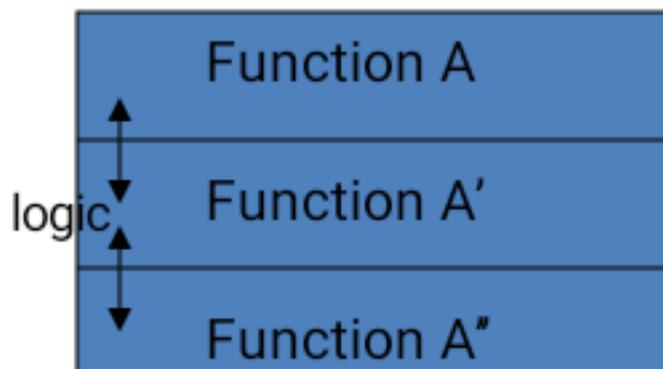
- Def.
 - module performs exactly one action
- Examples
 - get temperature of day
 - compute orbital of electron
 - calculate sales commission
- Why is this good?
 - more reusable
 - corrective maintenance easier
 - fault isolation
 - reduced regression faults
 - easier to extend product

Cohesion ratio = (number of modules having functional cohesion)/(total number of modules)

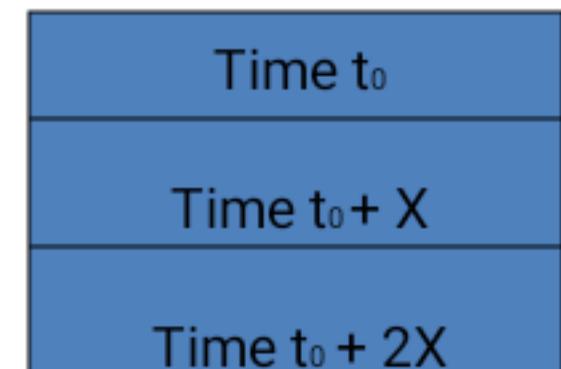
Examples of Cohesion-1



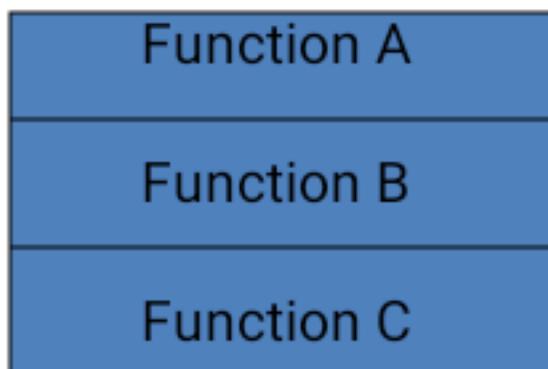
Coincidental
Parts unrelated



Logical
Similar functions

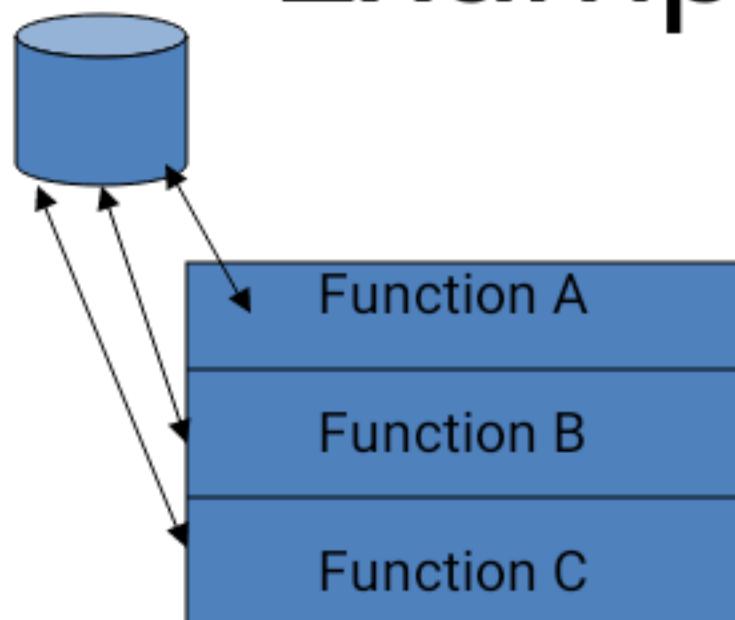


Temporal
Related by time

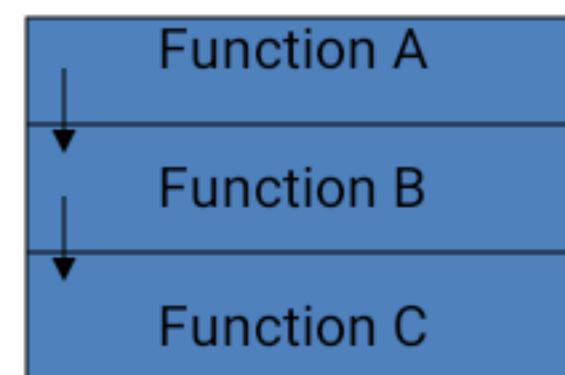


Procedural
Related by order of functions

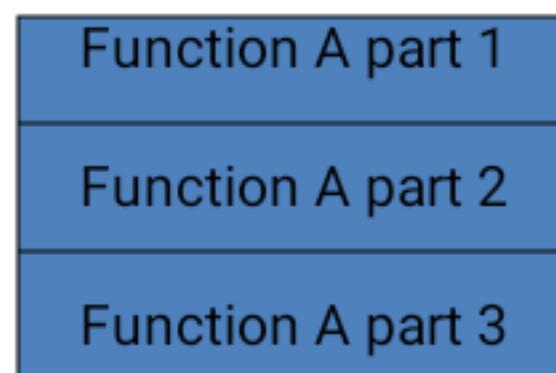
Examples of Cohesion-2



Communicational
Access same data



Sequential
Output of one is input to another

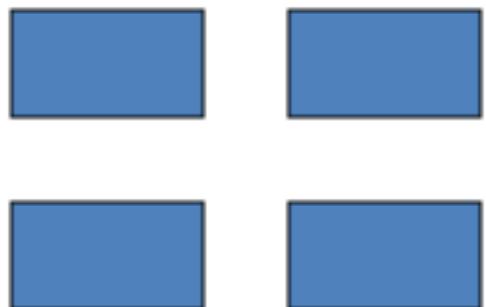


Functional
Sequential with complete, related functions

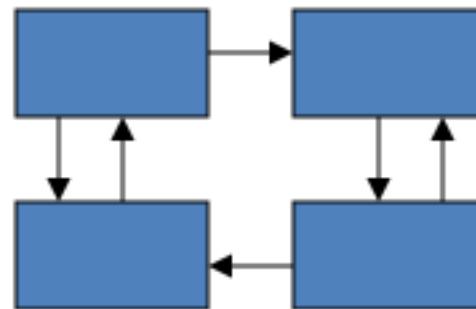
Study

- P1: What is the effect of cohesion on maintenance?
- P2: What is the effect of coupling on maintenance?
- P3: Produce an example of each type of cohesion. Justify your answers.
- P4: Produce an example of each type of coupling. Justify your answers.

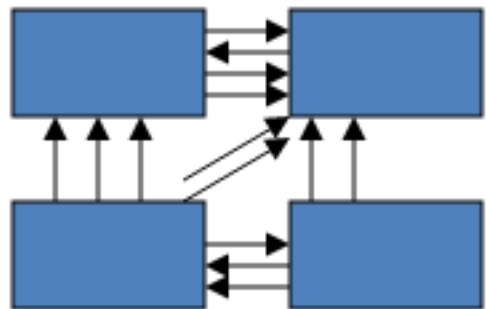
Coupling: Degree of dependence among components



No dependencies



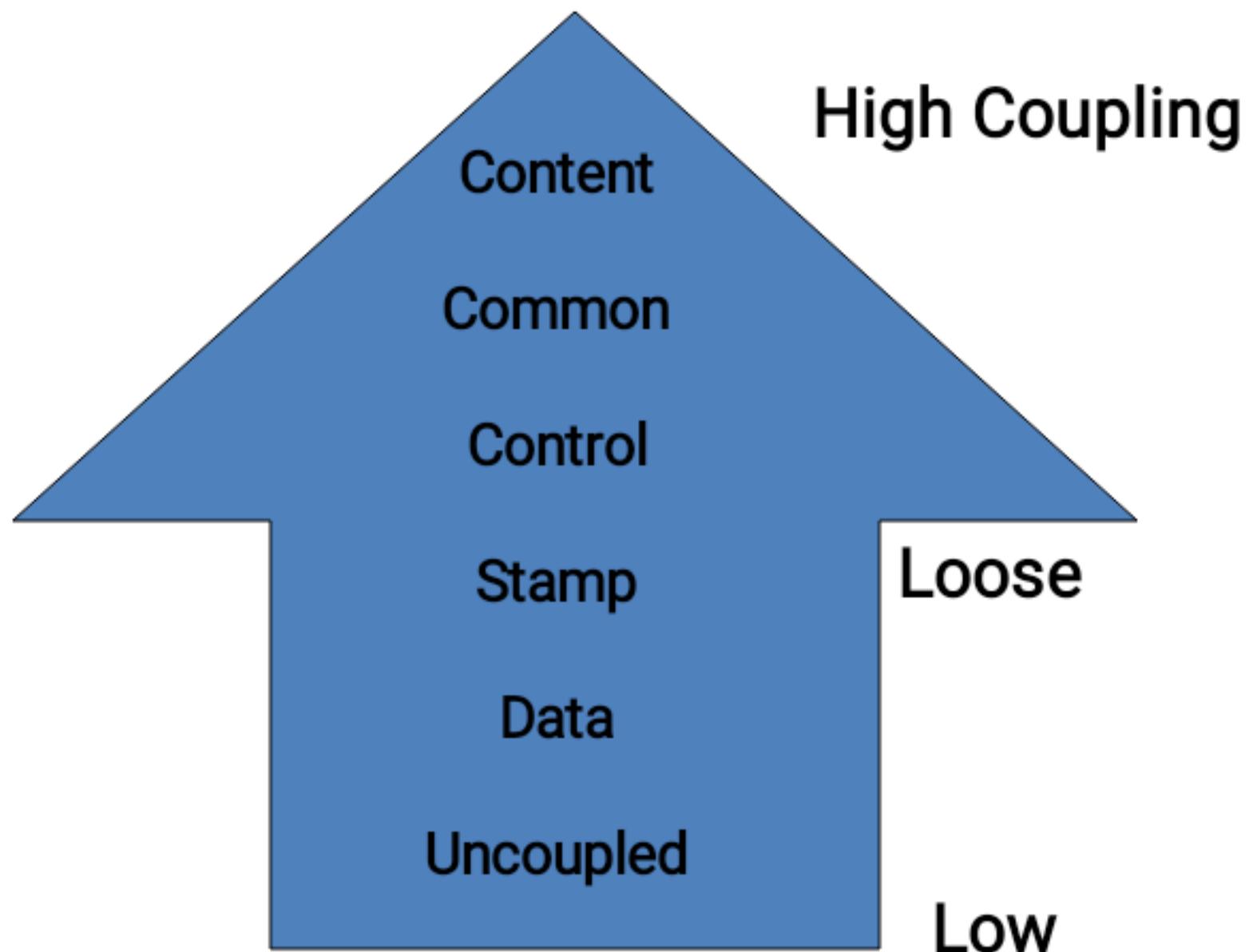
Loosely coupled-some dependencies



Highly coupled-many dependencies

High coupling makes modifying parts of the system difficult, e.g., modifying a component affects all the components to which the component is connected.

Range of Coupling



Tight coupling

- // Java program to illustrate
- // tight coupling concept
- class Subject {
- Topic t = new Topic();
- public void startReading()
- {
- t.understand();
- }
- }
- class Topic {
- public void understand()
- {
- System.out.println("Tight coupling concept");
- }
- }

Loose Coupling

```
public interface Topic
{
    void understand();
}

class Topic1 implements Topic {
    public void understand()
    {
        System.out.println("Got it");
    }
}

class Topic2 implements Topic {
    public void understand()
    {
        System.out.println("understand");
    }
}

public class Subject {
    public static void main(String[] args)
    {
        Topic t = new Topic1();
        t.understand();
    }
}
```

1. Content Coupling

- Def.
 - one module directly references contents of the other
 - x refers to the inside of y; i.e. it branches into, changes data in, or alters a statement in y.
- Example
 - module **a** modifies statements of module **b**
 - module **a** refers to local data of module **b** in terms of some numerical displacement within **b**
 - module **a** branches into local label of module **b**
- Why is this bad?
 - almost any change to **b** requires changes to **a**

Example of Content Coupling-2

Part of program handles lookup for customer.
When customer not found, component adds
customer by directly modifying the contents of
the data structure containing customer data.

Improvement:

When customer not found, component calls the
AddCustomer() method that is responsible for
maintaining customer data.

2. Common Coupling

- Def.
 - two modules have write access to the same global data
 - Undesirable; if the format of the global data must be changed, then all common-coupled modules must also be changed.
- Example
 - two modules have access to same database, and can both read and write same record
 - use of Java **public** statement
- Why is this bad?
 - resulting code is unreadable
 - modules can have side-effects
 - must read entire module to understand
 - difficult to reuse
 - module exposed to more data than necessary

Example-2

Process control component maintains current data about state of operation. Gets data from multiple sources. Supplies data to multiple sinks.

Each source process writes directly to global data store. Each sink process reads directly from global data store.

Improvement

Data manager component is responsible for data in data store. Processes send data to and request data from data manager.

3. Control Coupling

- Def.
 - one module passes an element of control to the other
 - x passes a parameter to y with the intention of controlling its behavior; that is, the parameter is a flag.
- Example
 - control-switch passed as an argument
- Why is this bad?
 - modules are not independent
 - module **b** must know the internal structure of module **a**
 - affects reusability

Example -2

- **Acceptable:** Module p calls module q and q passes back flag that says it cannot complete the task, then q is passing data
- **Not Acceptable:** Module p calls module q and q passes back flag that says it cannot complete the task and, as a result, writes a specific message.

4. Stamp Coupling

- Def.
 - data structure is passed as parameter, but called module operates on only some of individual components
 - x and y accept the same record type as a parameter. This type of coupling may cause interdependency between otherwise-unrelated modules.
- Example
 - **calculate withholding (employee record)**
- Why is this bad?
 - affects understanding
 - not clear, without reading entire module, which fields of record are accessed or changed
 - unlikely to be reusable
 - other products have to use the same higher level data structures
 - passes more data than necessary
 - e.g., uncontrolled data access can lead to computer crime

Example-2

Customer Billing System

The print routine of the customer billing accepts a customer data structure as an argument, parses it, and prints the name, address, and billing information.

Improvement

The print routine takes the customer name, address, and billing information as an argument.

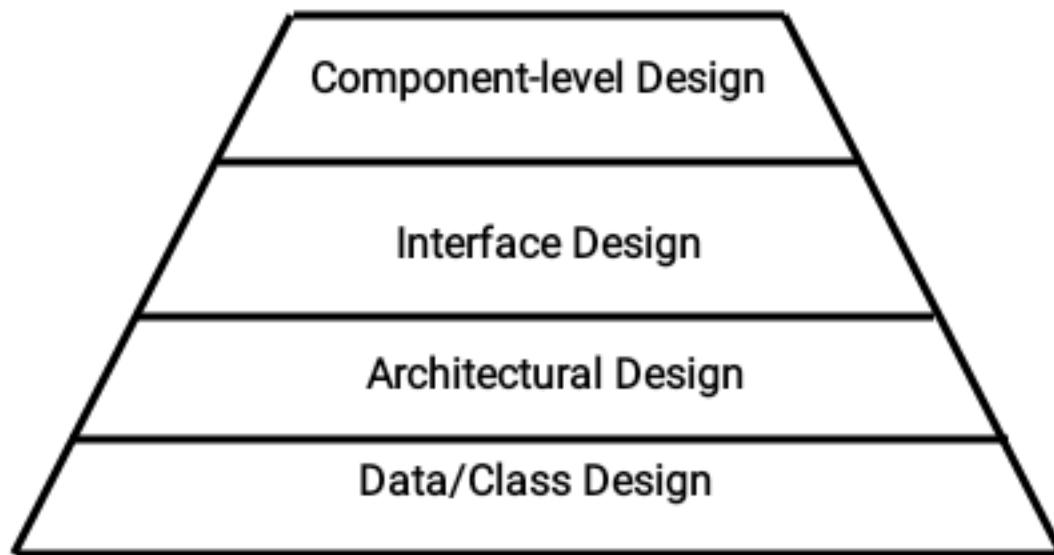
5. Data Coupling

- Def.
 - every argument is either a simple argument or a data structure in which all elements are used by the called module
- Example
 - display time of arrival (flight number)
 - get job with highest priority (job queue)
- Slippery slope between stamp & data (see queue)
- Why is this good?
 - maintenance is easier
 - good design has high cohesion & weak coupling

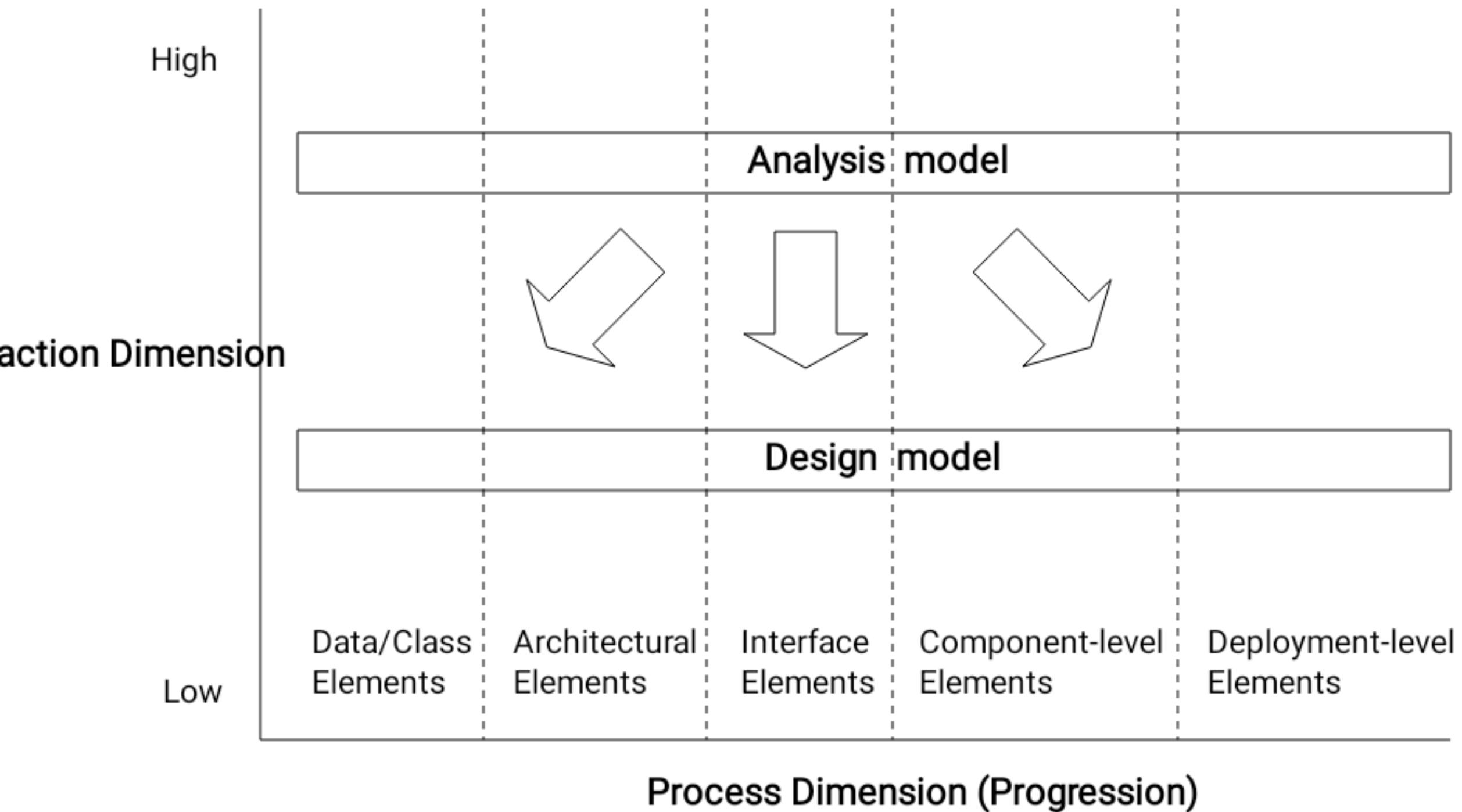
6. No Coupling

- x and y have no communication; that is, they are totally independent of one another.

The Design Model



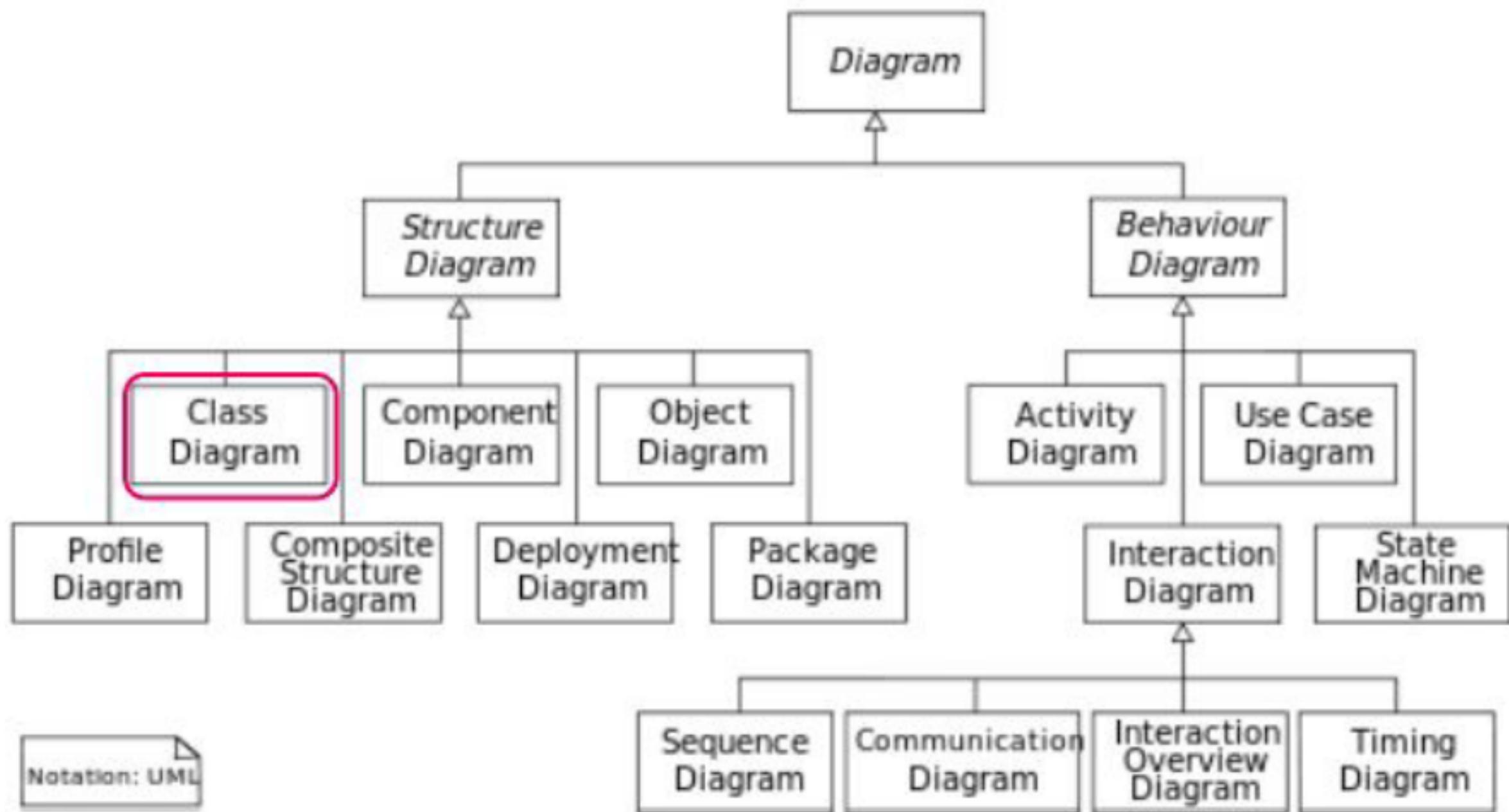
Dimensions of the Design Model



UML class diagram

The class notes are a compilation and edition from many sources for which the references are given at the end. Thanks to them!

UML Diagram Types



UML class diagrams

- What is a UML class diagram?
 - **UML class diagram:** a picture of the classes in an OO system, their fields and methods, and connections between the classes that interact or inherit from each other
- What are some things that are not represented in a UML class diagram?
 - details of how the classes interact with each other
 - algorithmic details; how a particular behavior is implemented

Class diagram

- A class diagram depicts classes and their interrelationships
- Used for describing **structure and behavior** in the use cases
- Provide a conceptual model of the system in terms of entities and their relationships
- Used for requirement capture, end-user interaction
- Detailed class diagrams are used for developers

Class diagram

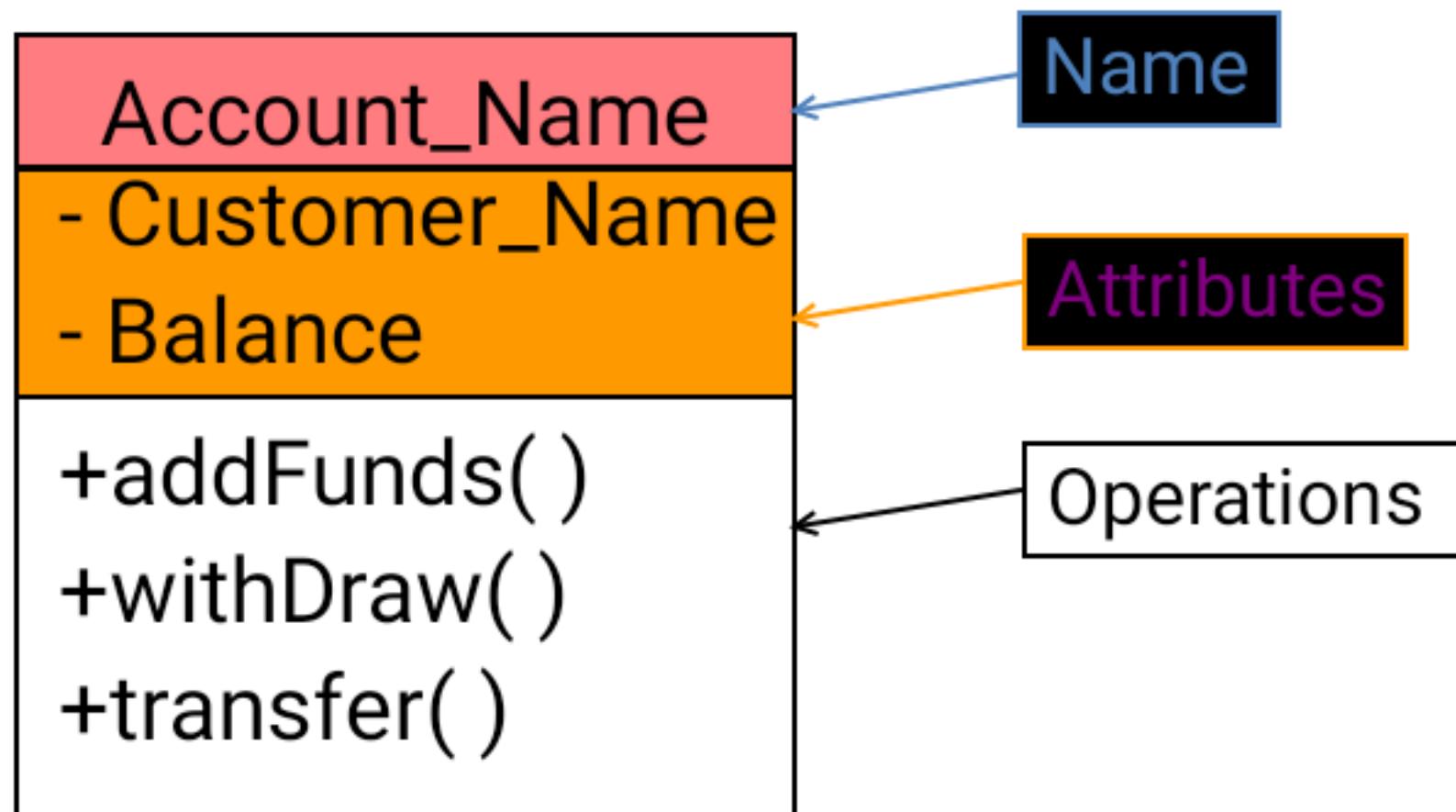
- Each class is represented by a rectangle subdivided into three compartments
 - ★ Name
 - ★ Attributes
 - ★ Operations
- Modifiers are used to indicate visibility of attributes and operations.
 - ★ '+' is used to denote *Public* visibility (everyone)
 - ★ '#' is used to denote *Protected* visibility (friends and derived)
 - ★ '-' is used to denote *Private* visibility (no one)
- By default, attributes are hidden and operations are visible.

Access Specifiers

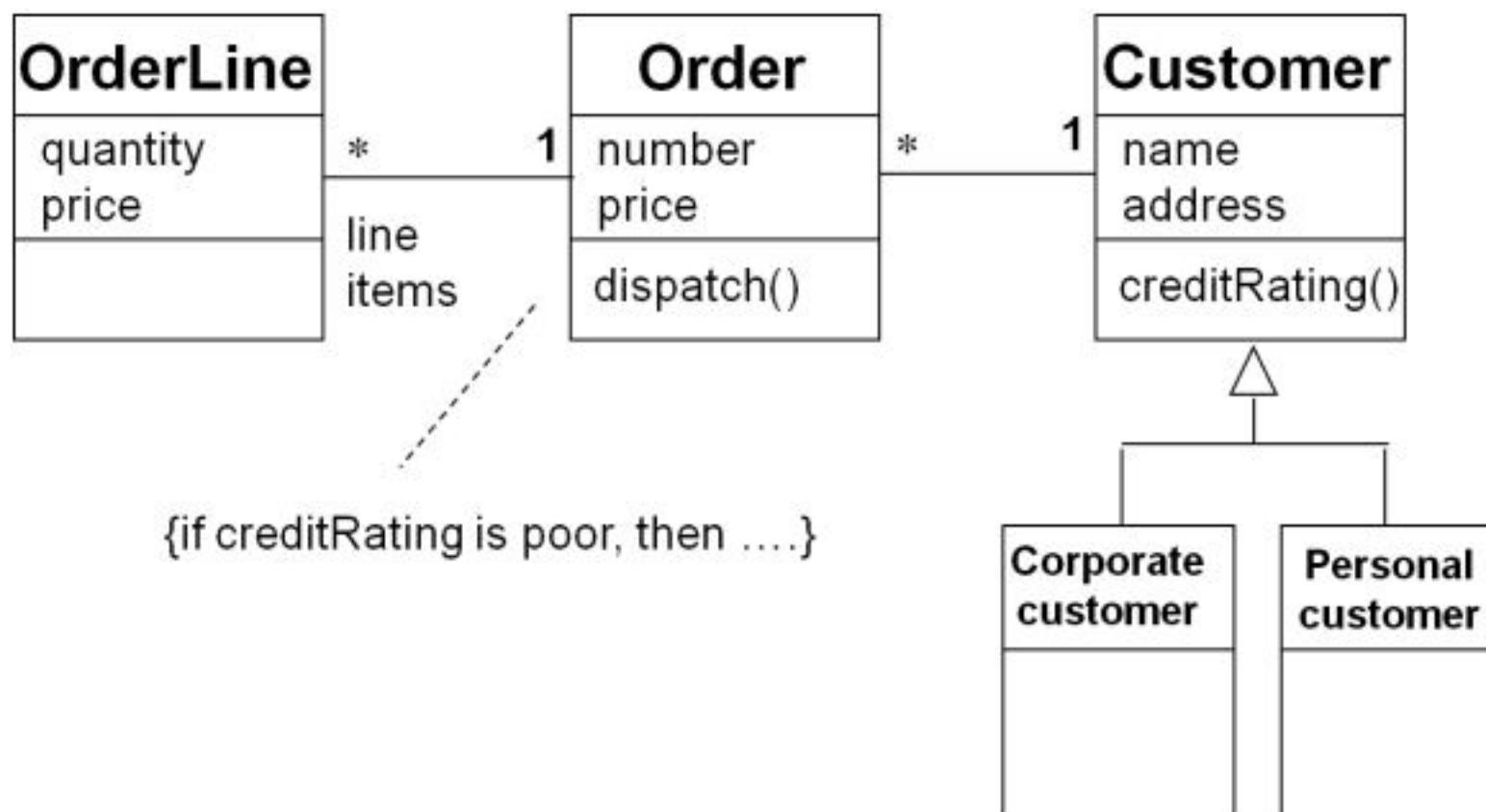
Access for each of these visibility types is shown below for members of different classes

Access	public (+)	private (-)	protected (#)
Members of the same class	yes	yes	yes
Members of derived classes	yes	no	yes
Members of any other class	yes	no	no

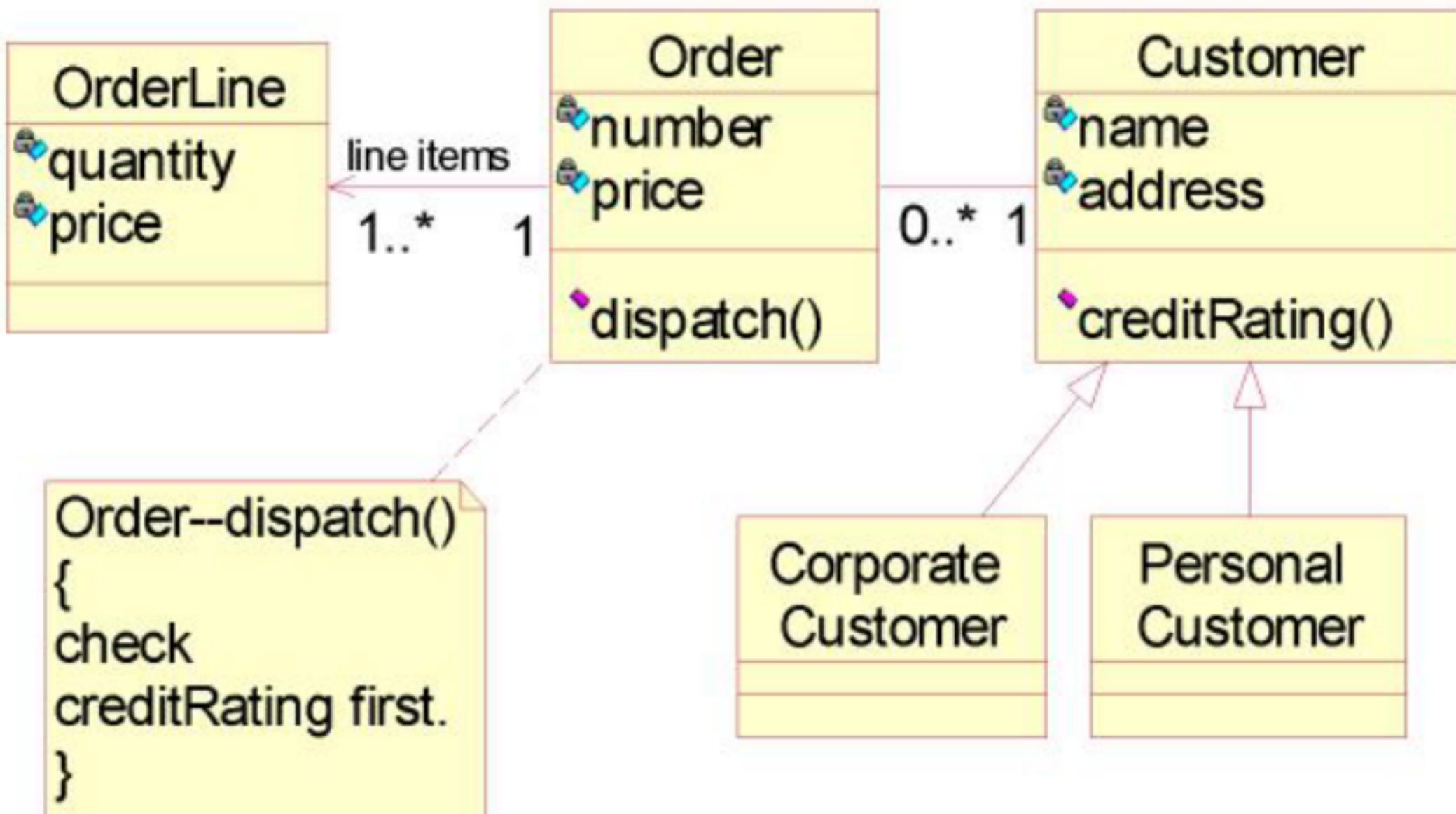
Class diagram



Class Diagram: Example



Rational Rose - Example of a class diagram



Attributes

- Attributes represent **characteristics** or **properties** of objects
- They are place holders or slots that hold values
- The values they hold are other objects
- The name of an attribute communicates its meaning
- An attribute can be defined for individual objects or classes of objects
 - If defined for a class, then every object in the class has that attribute (place holder)

Class Attributes (Cont'd)

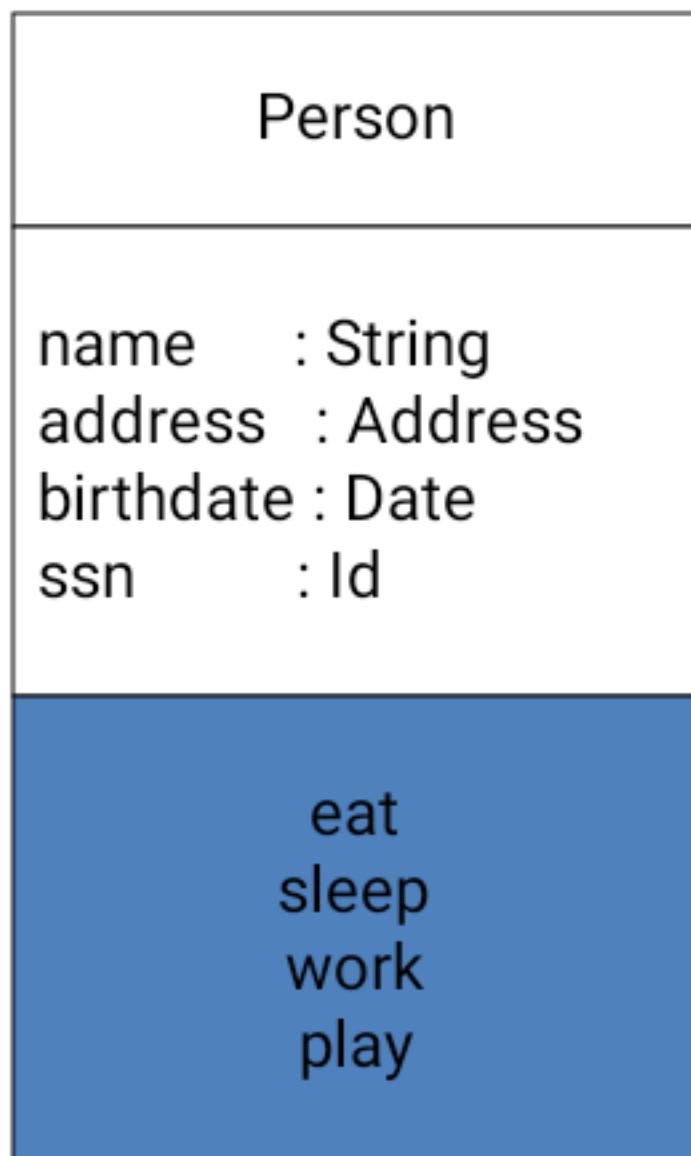
Person

```
+ name    : String  
# address : Address  
# birthdate : Date  
/ age     : Date  
- ssn     : Id
```

Attributes can be:

- + public
- # protected
- private
- / derived

Class Operations



Operations describe the class behavior and appear in the third compartment.

Class Names

- The name should be a **noun or noun phrase**
- The name should be singular and description of each object in the class
- The name should be meaningful from a problem-domain perspective
 - “Student” is better than “Student Data” or “S-record” or any other implementation driven name
- Avoid **jargon** in the names
- Try to make the name descriptive of the class’s common properties

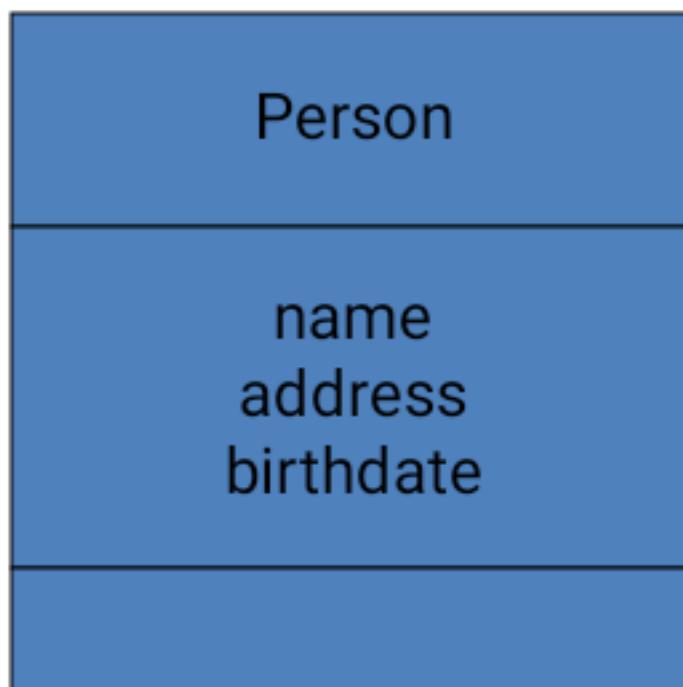
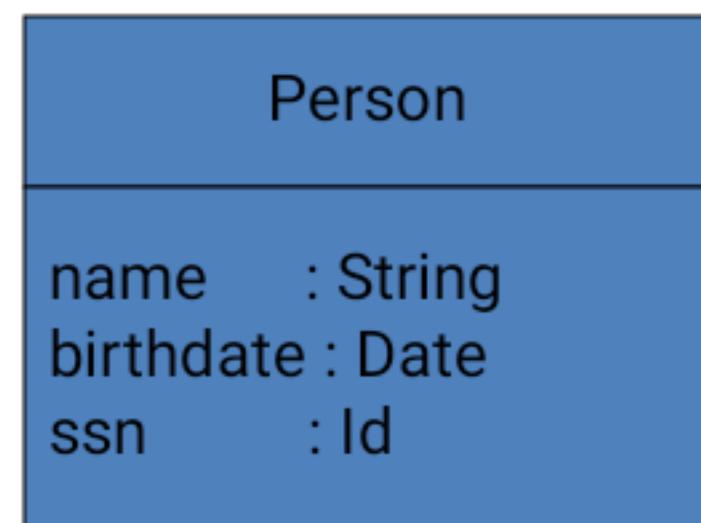
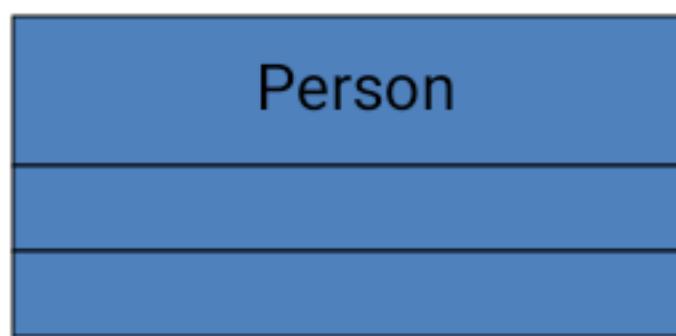
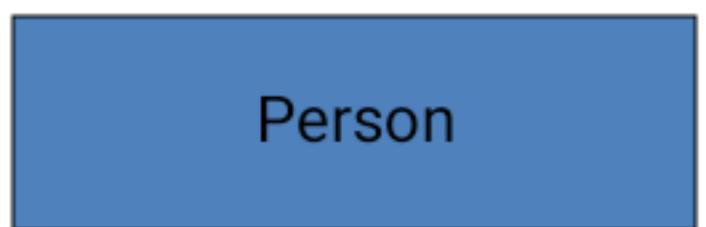
Exercise – Class Identification

- Identify meaningful classes and attributes in the:

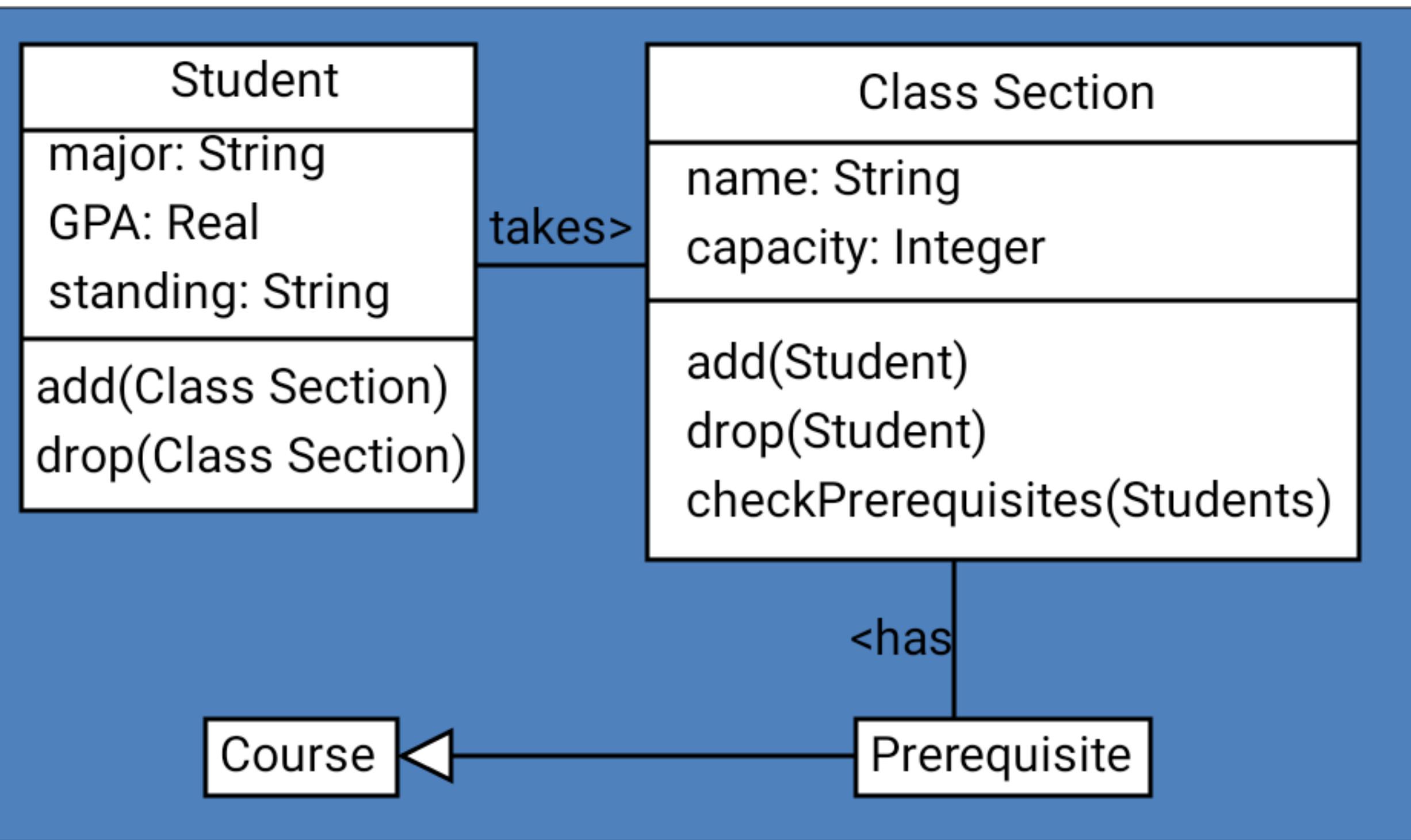
SafeHome System

Depicting Classes

When drawing a class, you needn't show attributes and operation in every diagram.



Operations

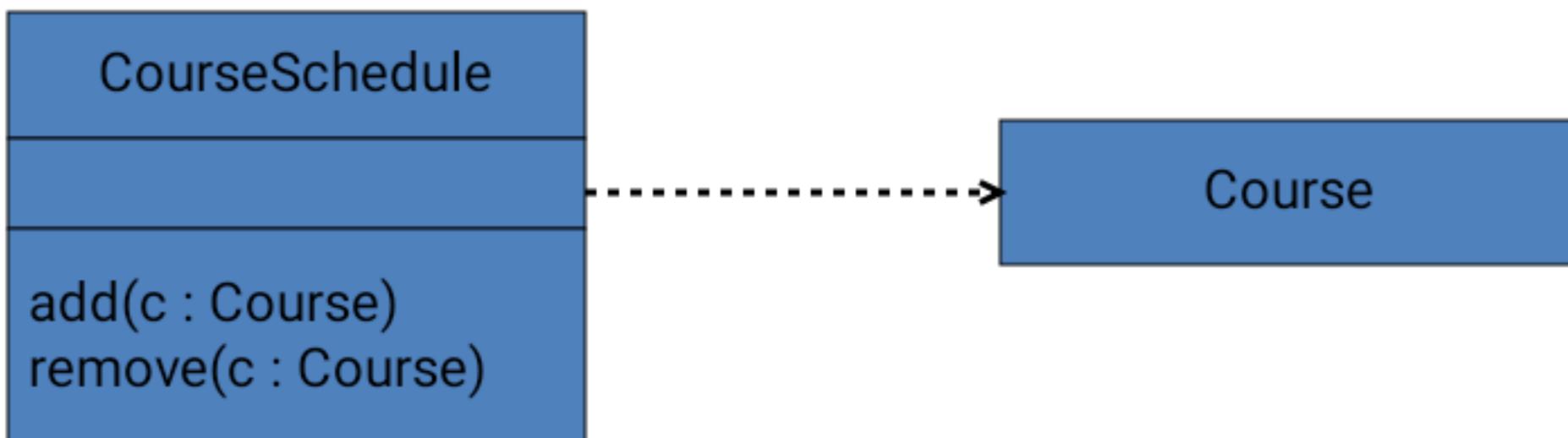


OO Relationships

- There are two kinds of Relationships
 - ★ Dependencies
 - ★ Generalization (parent-child relationship)
 - ★ Association (student enrolls in course)
- Associations can be further classified as
 - ★ Aggregation
 - ★ Composition

Dependency Relationships

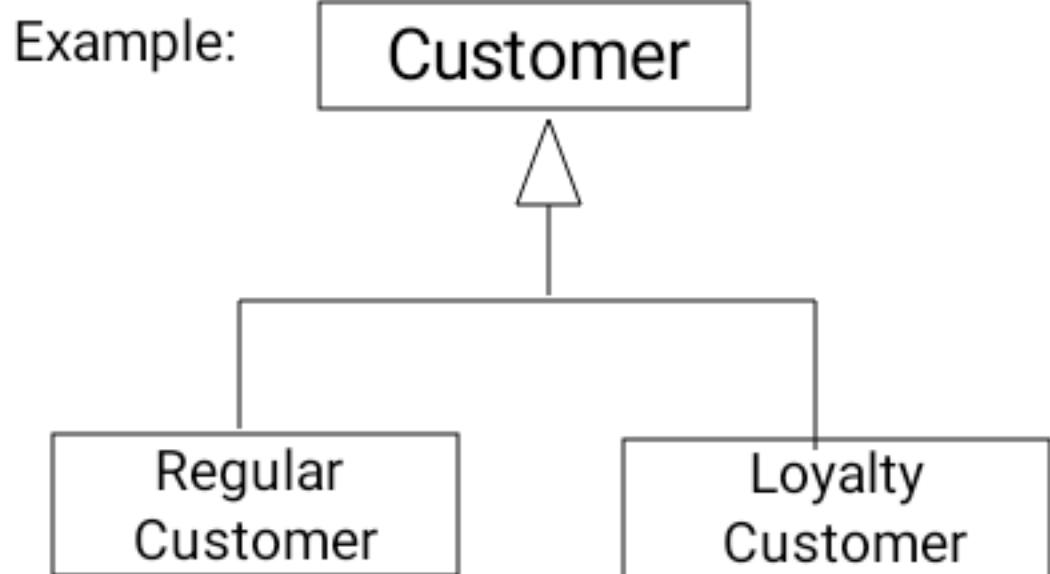
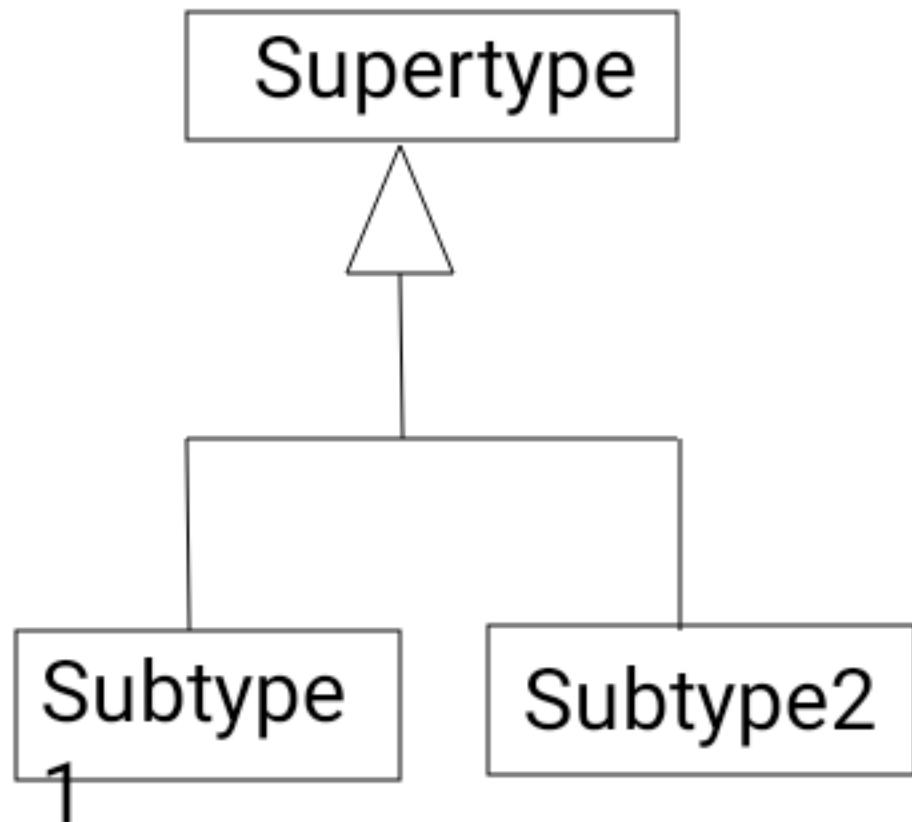
A *dependency* indicates a semantic relationship between two or more elements. The dependency from *CourseSchedule* to *Course* exists because *Course* is used in both the **add** and **remove** operations of *CourseSchedule*.



Dependencies

- Syntax:
 - a dashed link with an straight-line arrowhead point to a component on which there is a dependency
- Dependencies can be defined among: classes, notes, packages, and other types of components
- Can dependencies go both ways?
- Any problems with having lots of dependencies?

OO Relationships: Generalization

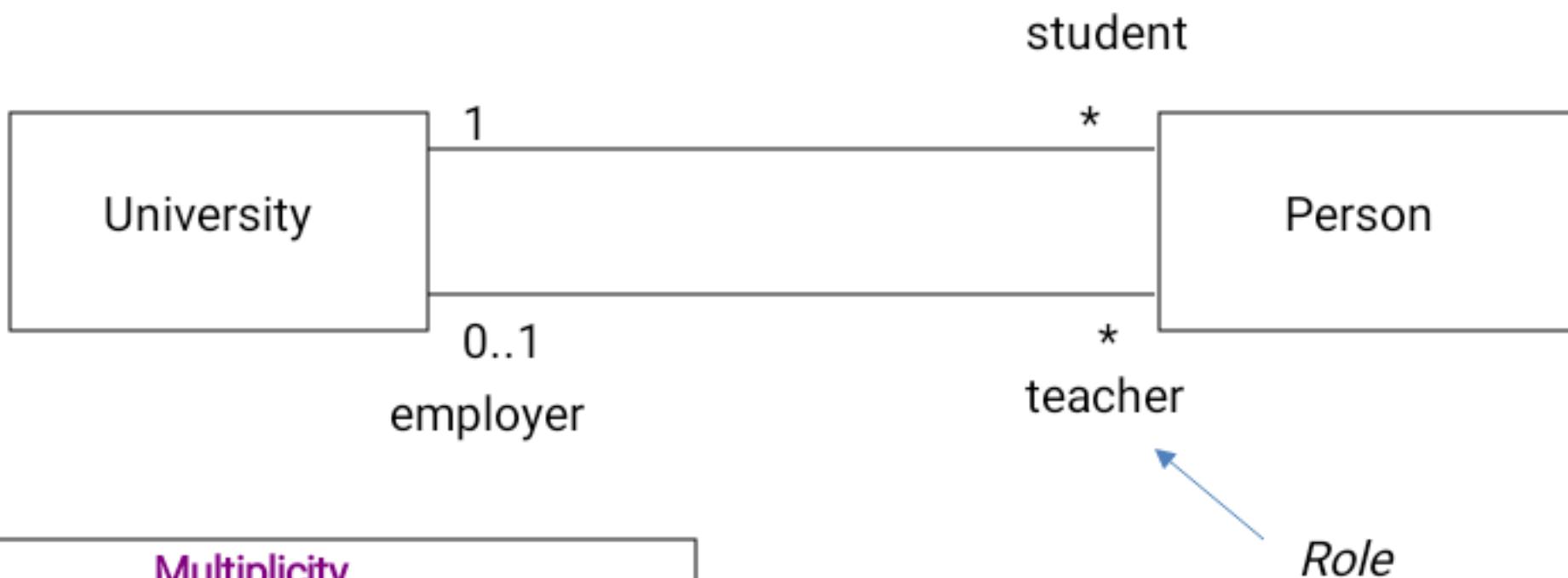


- Inheritance is a required feature of object orientation
- Generalization expresses a parent/child relationship among related classes.
- Used for abstracting details in several layers

OO Relationships: Association

- Represent relationship between instances of classes
 - ★ Student enrolls in a course
 - ★ Courses have students
 - ★ Courses have exams
 - ★ Etc.
- Association has two ends
 - ★ Role names (e.g. enrolls)
 - ★ Multiplicity (e.g. One course can have many students)
 - ★ Navigability (unidirectional, bidirectional)

Association: Multiplicity and Roles



Multiplicity

Symbol Meaning

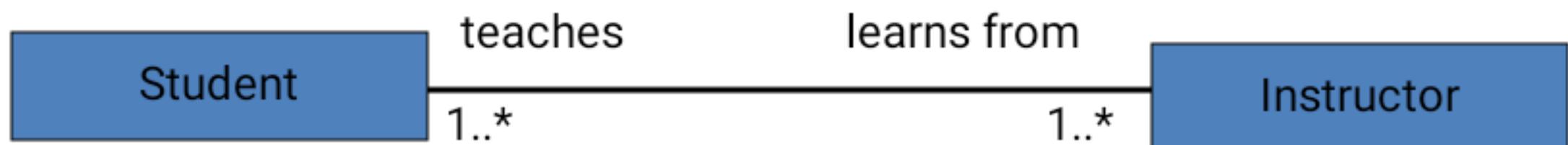
- 1 One and only one
- 0..1 Zero or one
- M..N From M to N (natural language)
- * From zero to any positive integer
- 0..* From zero to any positive integer
- 1..* From one to any positive integer

Role

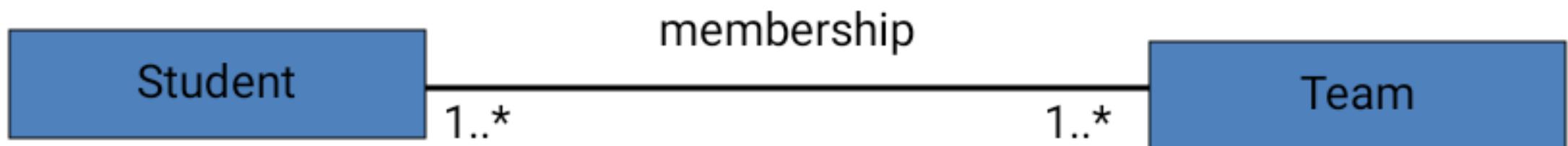
"A given university groups many people; some act as students, others as teachers. A given student belongs to a single university; a given teacher may or may not be working for the university at a particular time."

Association Relationships (Cont'd)

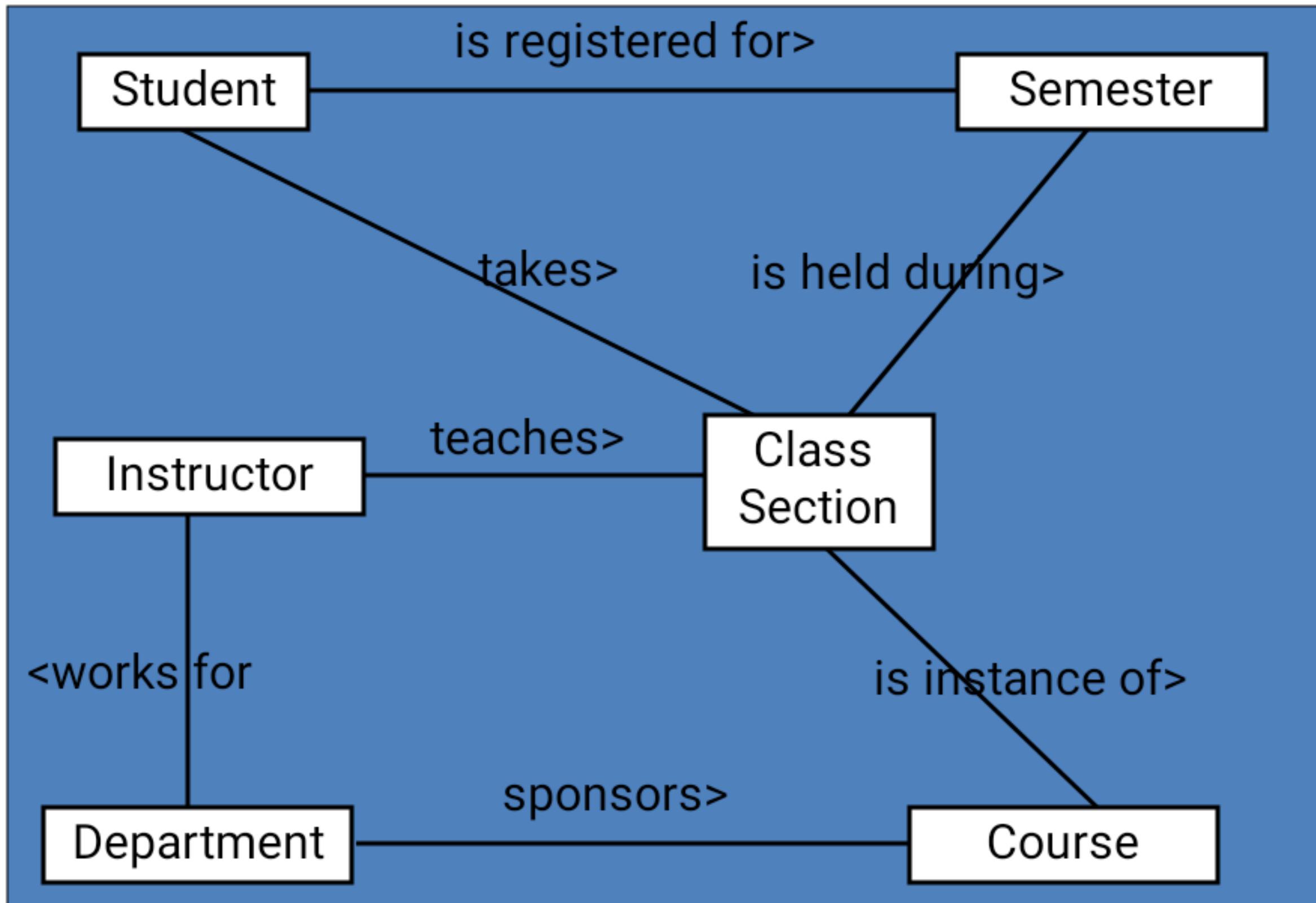
We can also indicate the behavior of an object in an association (*i.e.*, the *role* of an object) using *rolenames*.



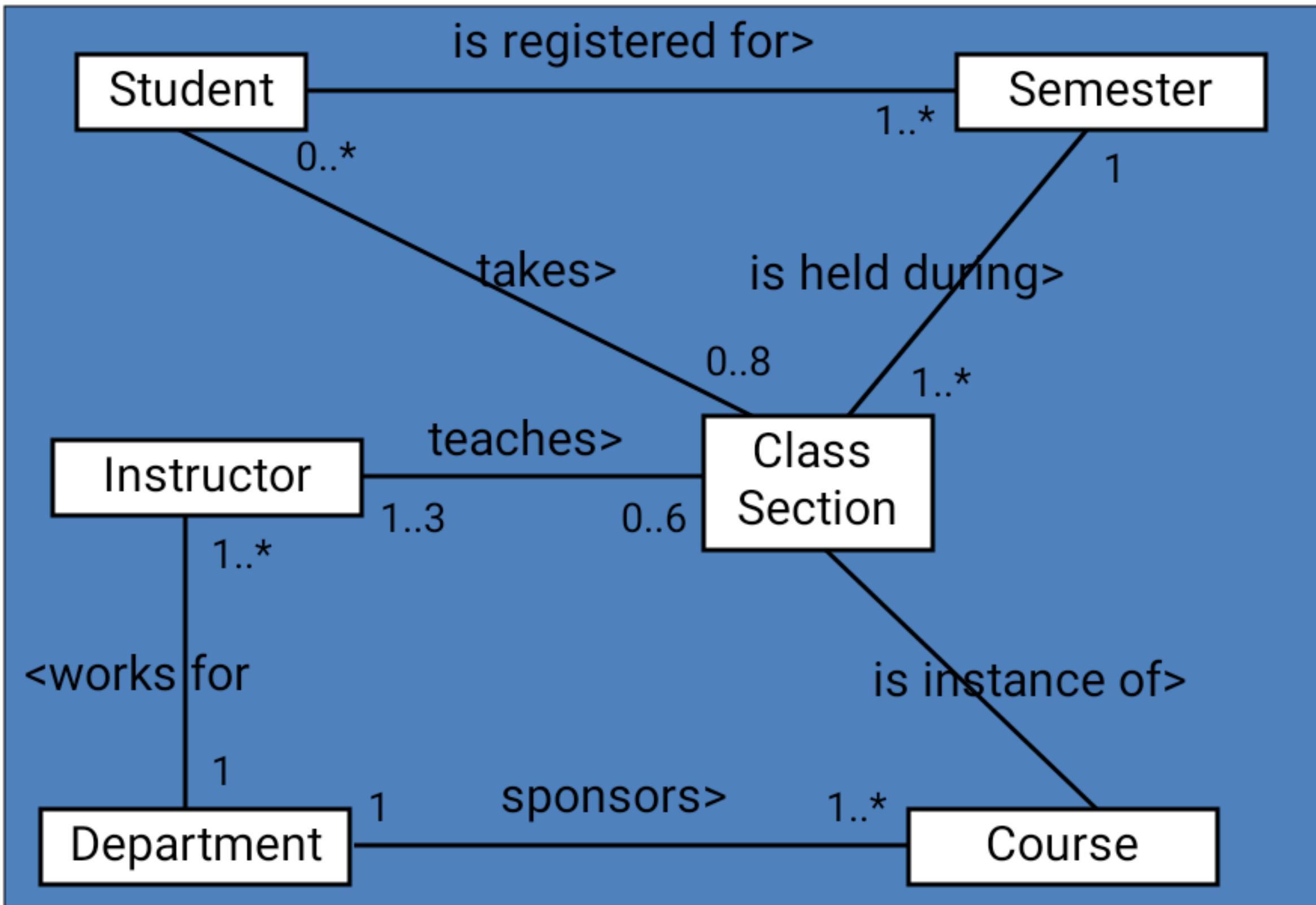
We can also name the association



Associations



Multiplicity Constraints



Questions

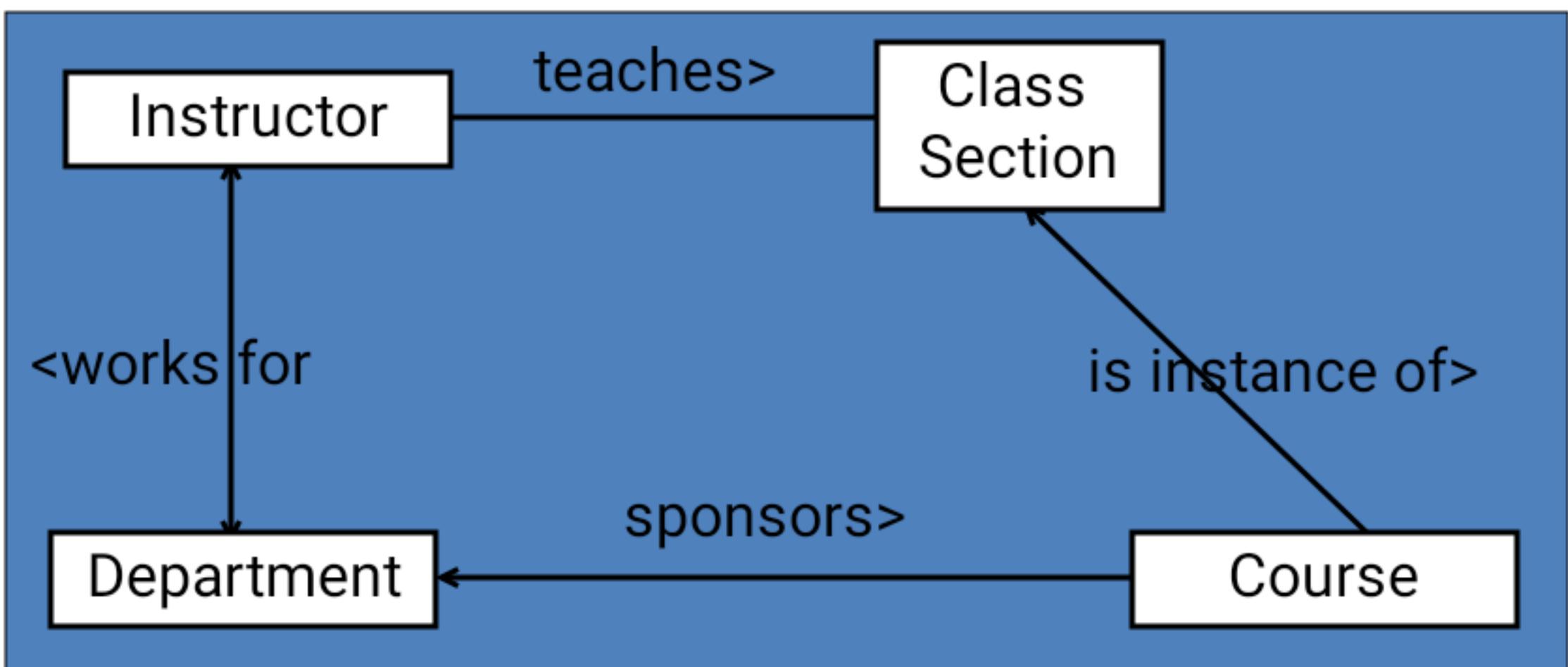
- From the previous diagram
 - How many classes can a student take?
 - Do you have to be registered in any classes to be a student?
 - Do I need to teach this class to be an Instructor?
 - Do I need to teach ANY classes?

Association Names

- Associations may be named
 - The names should communicate the meaning of the links
 - The names are typically verb phrases
 - The name should include an arrow indicating the direction in which the name should be read

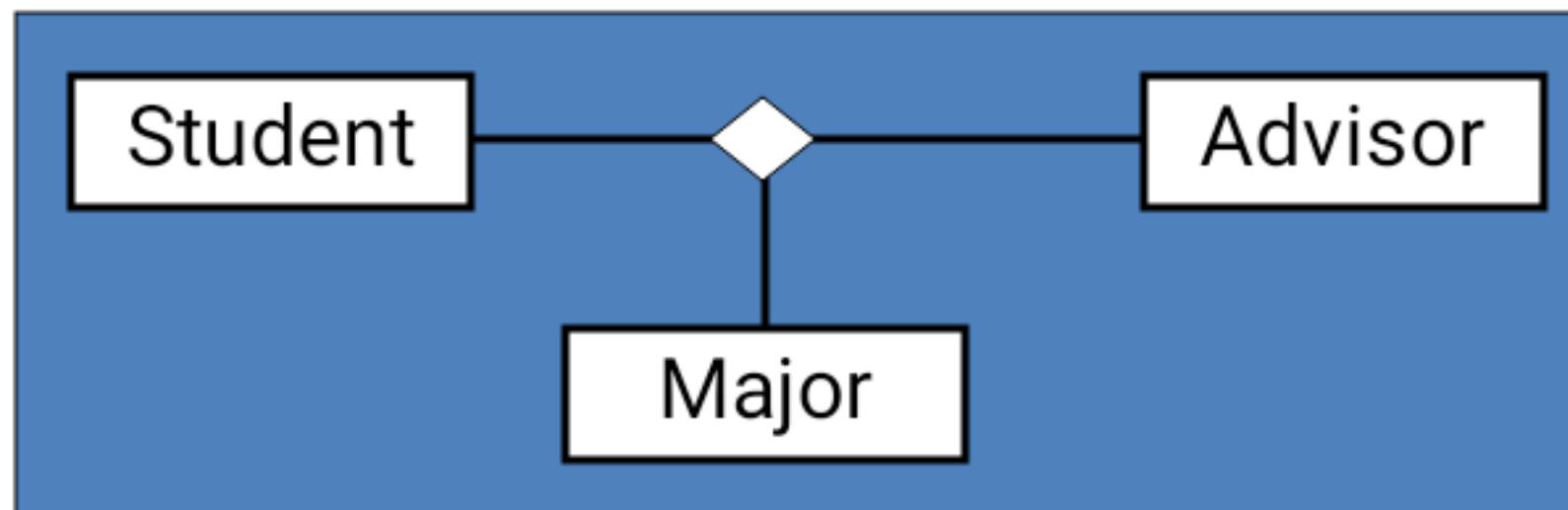
Navigation

- The navigation of associations can be
 - uni-directional
 - bi-directional
 - unspecified



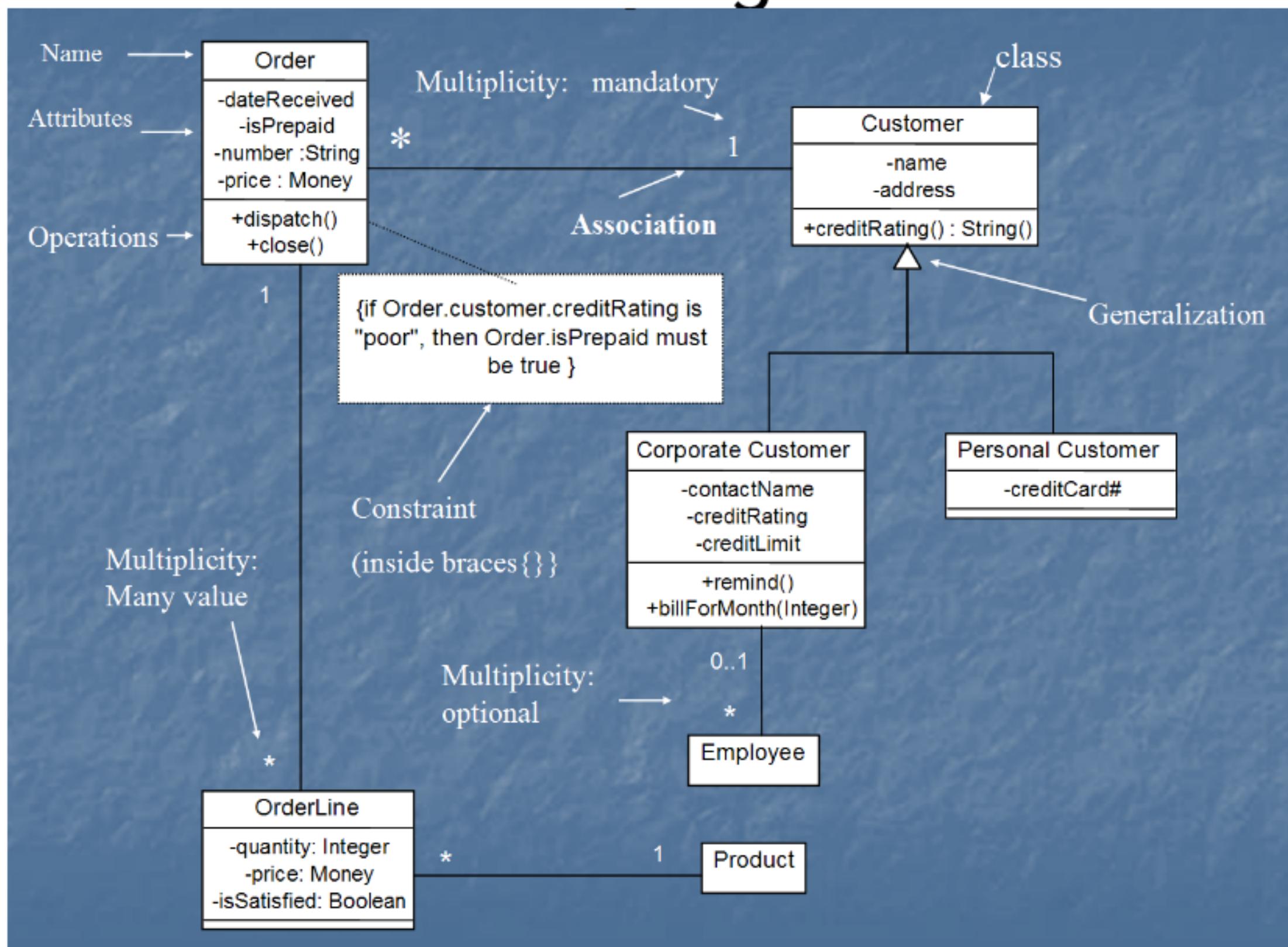
N-ary Associations

- Associations can connect more than one class
- Notation:



- How should we go about naming an n-ary association?

Class diagram



[from *UML Distilled Third Edition*]

Association: Model to Implementation

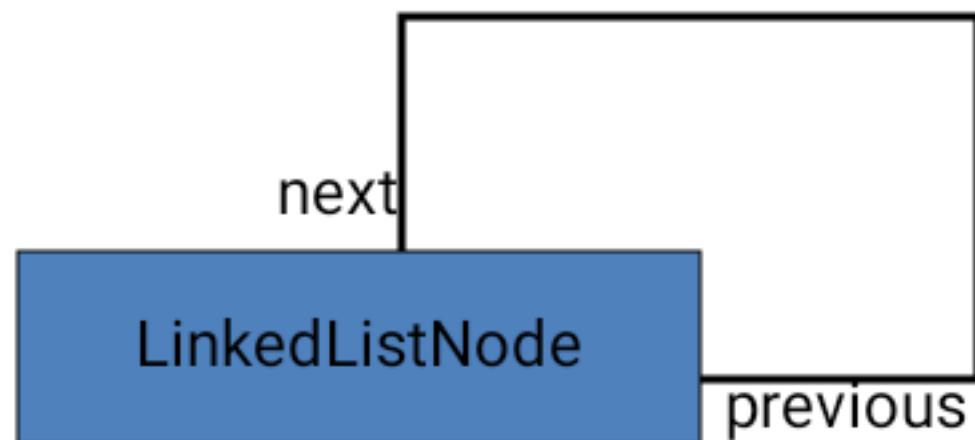


```
Class Student {  
    Course enrolls[4];  
}
```

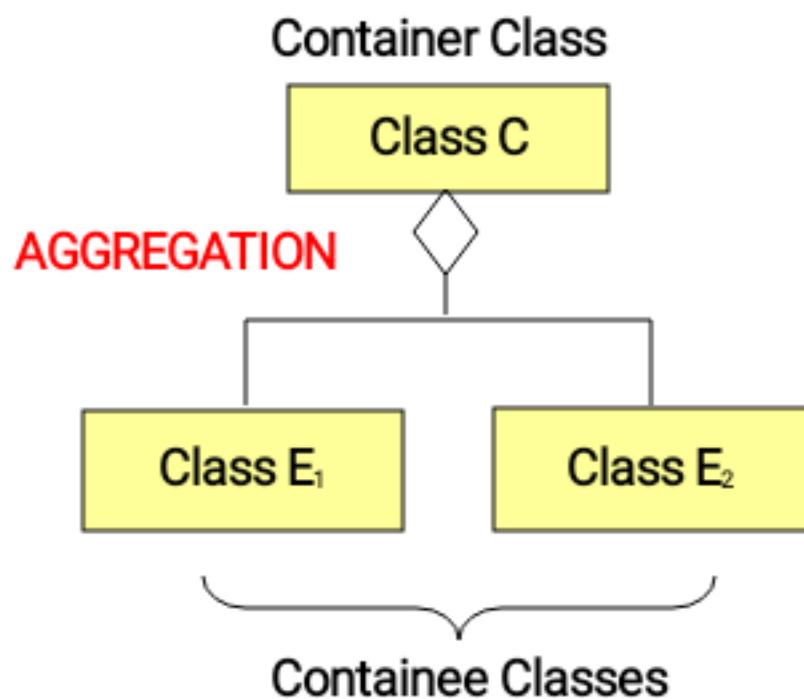
```
Class Course {  
    Student have[];  
}
```

Association Relationships (Cont'd)

A class can have a *self association*.



OO Relationships: Aggregation



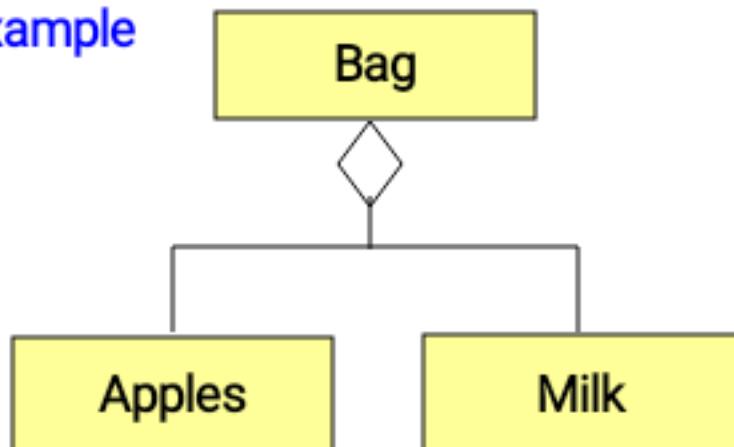
Aggregation:

expresses a relationship among instances of related classes. It is a specific kind of Container-Containee relationship.

express a more informal relationship than composition expresses.

Aggregation is appropriate when Container and Containees have no special access privileges to each other.

Example

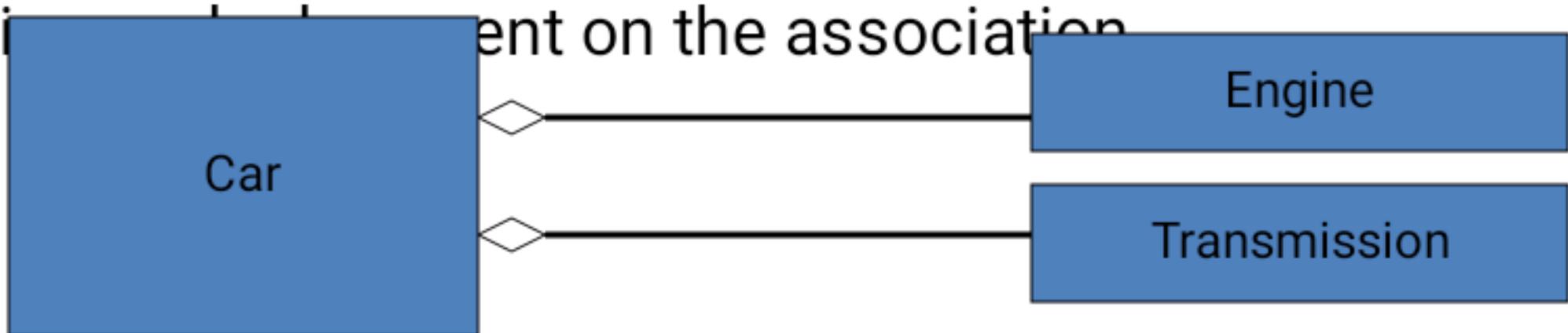


[From Dr.David A. Workman]

Aggregation

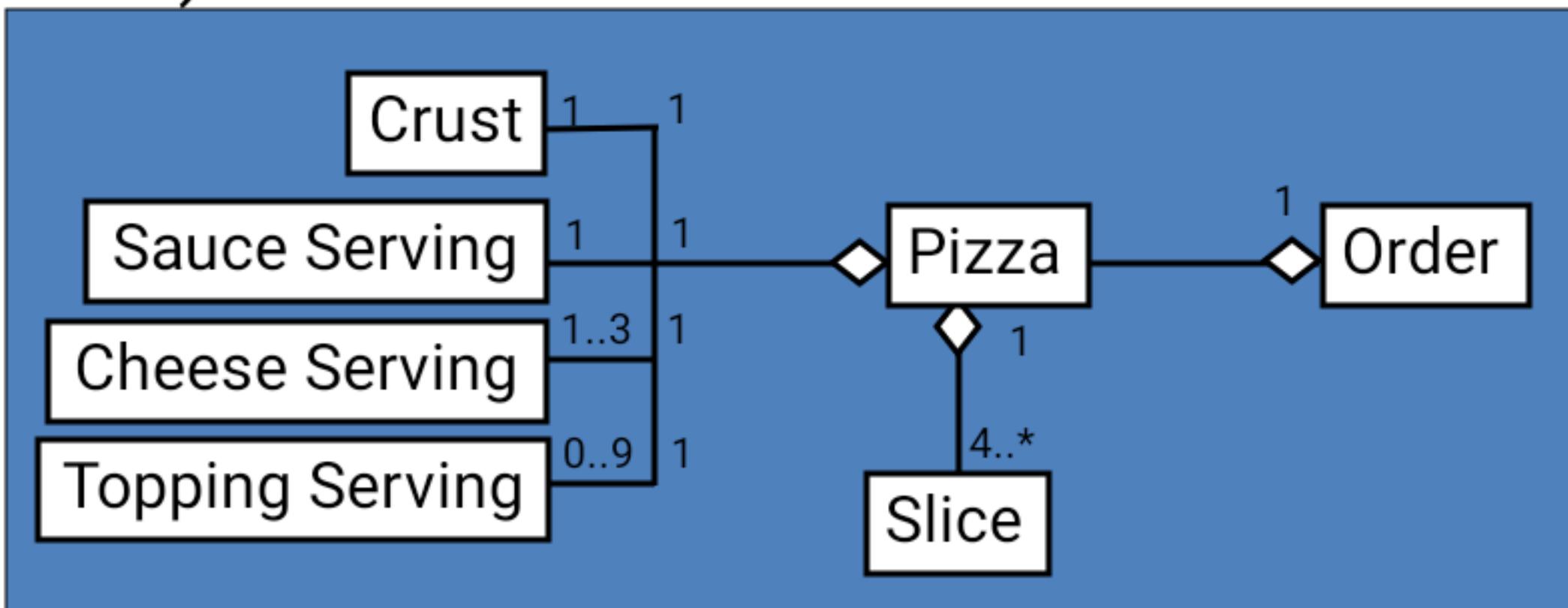
We can model objects that contain other objects by way of special associations called *aggregations* and *compositions*.

An *aggregation* specifies a whole-part relationship between an aggregate (a whole) and a constituent part, where the part can exist independently from the aggregate. Aggregations are denoted by a hollow-diamond symbol on the association line.



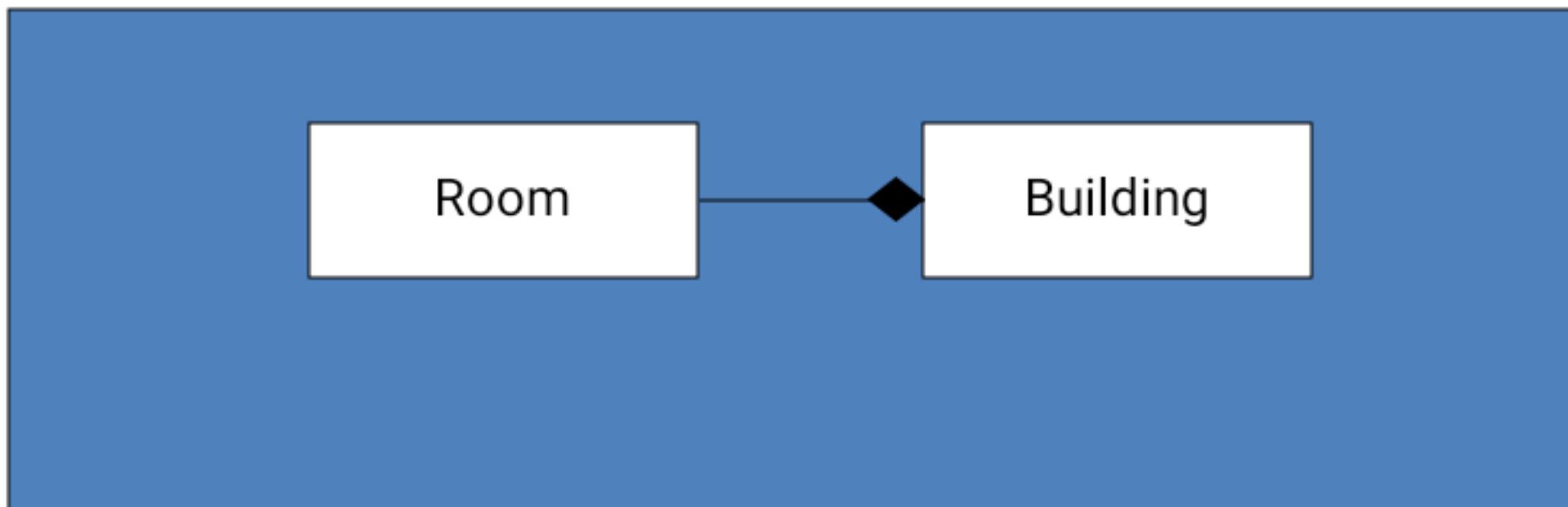
Aggregation

- *Aggregation*: is a special kind of association that means “part of”
- Aggregations should focus on single type of composition (physical, organization, etc.)



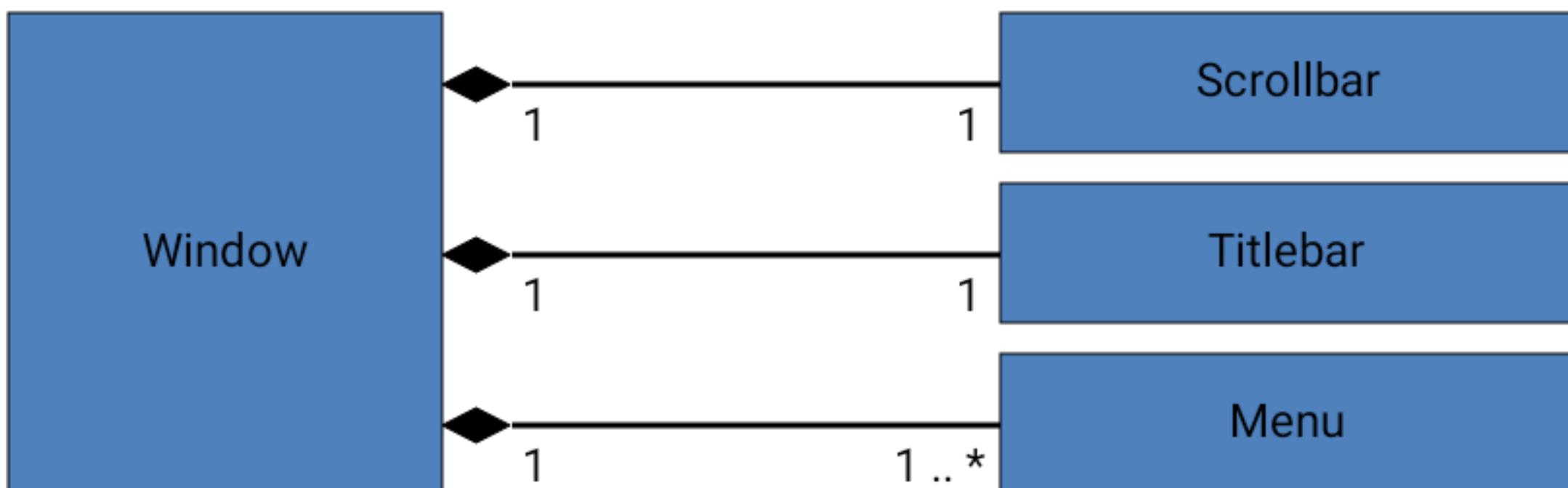
OO Relationships: Composition (very similar to aggregation)

- Think of composition as a stronger form of aggregation. Composition means something is a part of the whole, but cannot survive on its own.

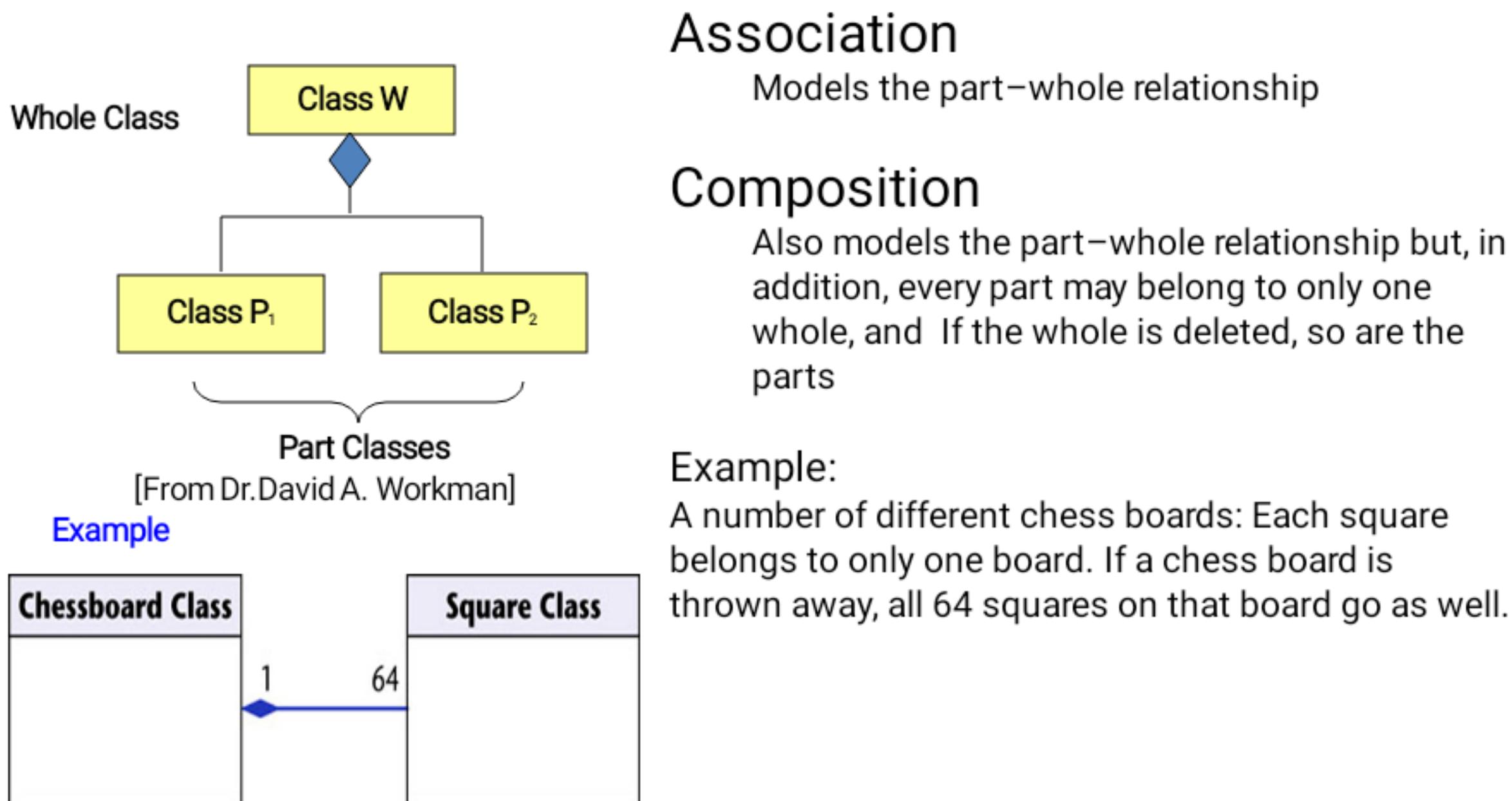


Composition

A *composition* indicates a strong ownership and coincident lifetime of parts by the whole (*i.e.*, they live and die as a whole). Compositions are denoted by a filled-diamond adornment on the association.



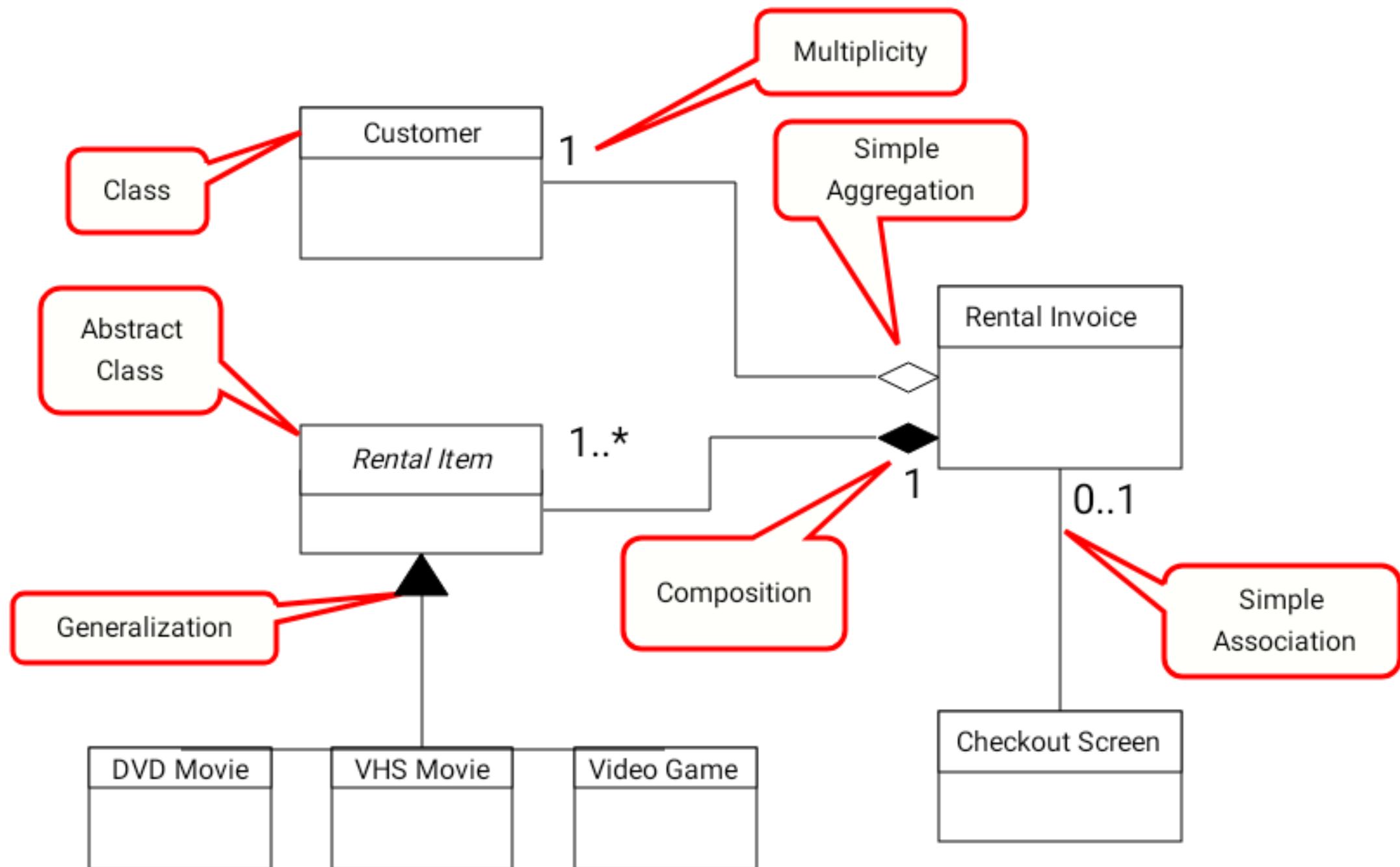
Composition



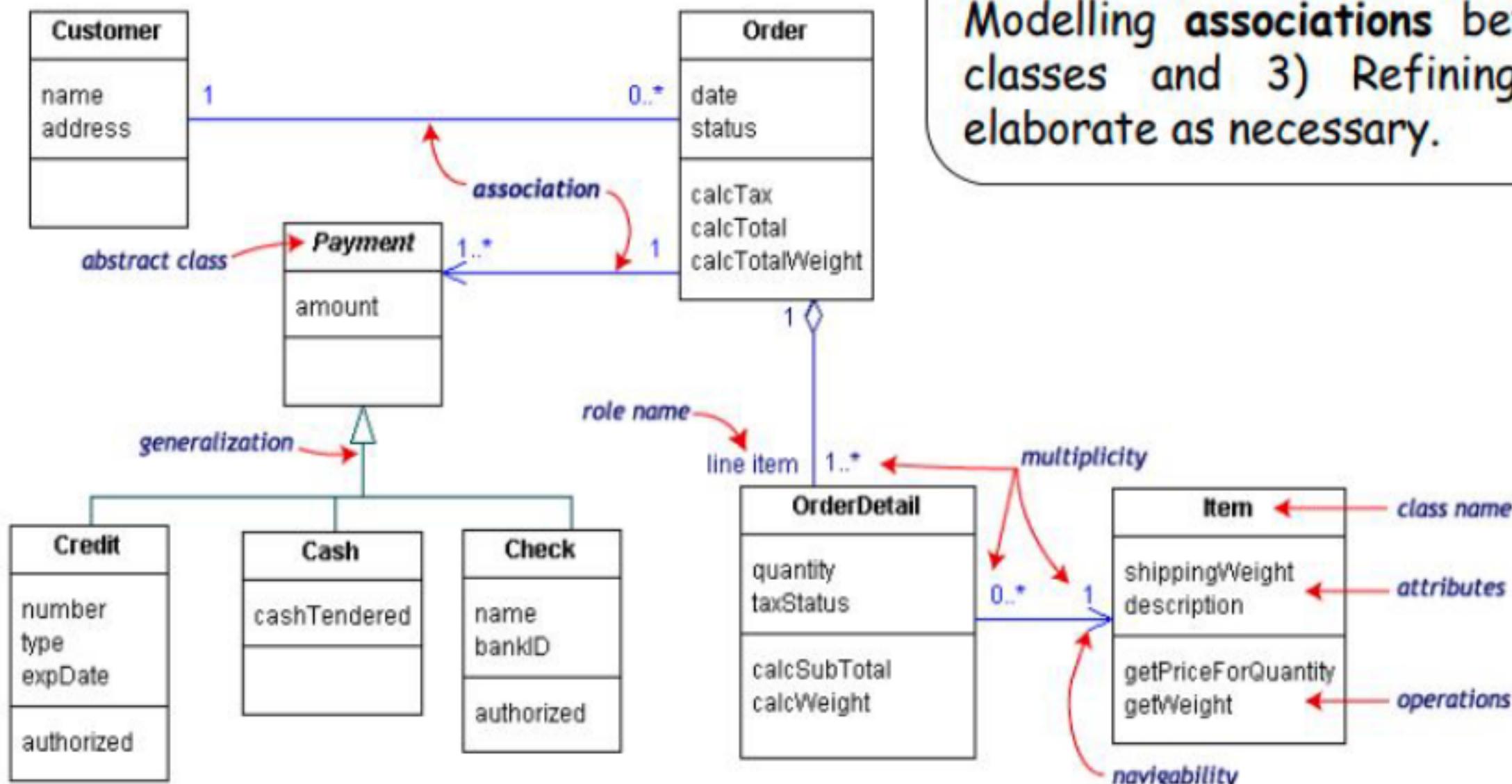
Aggregation vs. Composition

- **Composition** is really a strong form of **association**
 - ★ components have only one owner
 - ★ components cannot exist independent of their owner
 - ★ components live or die with their owner
 - ★ e.g. Each car has an engine that can not be shared with other cars.
- **Aggregations**
 - may form "part of" the association, but may not be essential to it.
They may also exist independent of the aggregate. e.g. Apples may exist independent of the bag.

Class diagram example 2

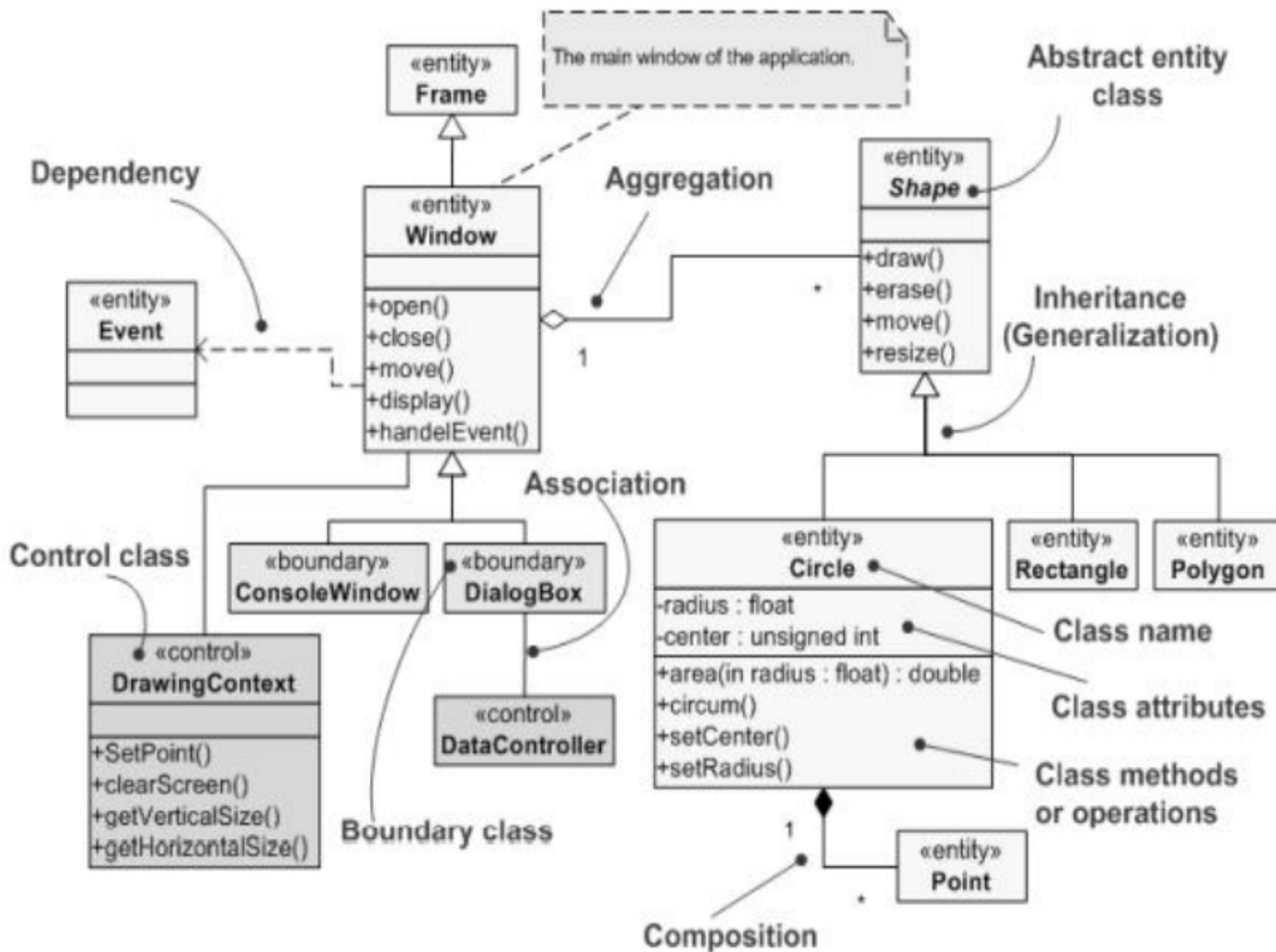


Class diagram example 3



These diagrams contain classes and associations. Construction involves: 1) Modelling **Classes**, 2) Modelling **associations** between classes and 3) Refining and elaborate as necessary.

Class diagram example 4



Class diagram example 4 - Observations

1. *Shape* is an abstract class. It is shown in Italics.
2. *Shape* is a superclass. *Circle*, *Rectangle* and *Polygon* are derived from *Shape*. In other words, a *Circle is-a Shape*. This is a generalization / inheritance relationship.
3. There is an association between *DialogBox* and *DataController*.
4. *Shape* is *part-of Window*. This is an aggregation relationship. *Shape* can exist without *Window*.
5. *Point* is *part-of Circle*. This is a composition relationship. *Point* cannot exist without a *Circle*.
6. *Window* is dependent on *Event*. However, *Event* is not dependent on *Window*.
7. The attributes of *Circle* are *radius* and *center*. This is an entity class.
8. The method names of *Circle* are *area()*, *circum()*, *setCenter()* and *setRadius()*.
9. The parameter *radius* in *Circle* is an *in* parameter of type *float*.
10. The method *area()* of class *Circle* returns a value of type *double*.
11. The attributes and method names of *Rectangle* are hidden. Some other classes in the diagram also have their attributes and method names hidden.

Interfaces

<<interface>>
ControlPanel

An *interface* is a named set of operations that specifies the behavior of objects without showing their inner structure.

It can be rendered in the model by a one- or two-compartment rectangle, with the *stereotype* <<interface>> above the interface name.

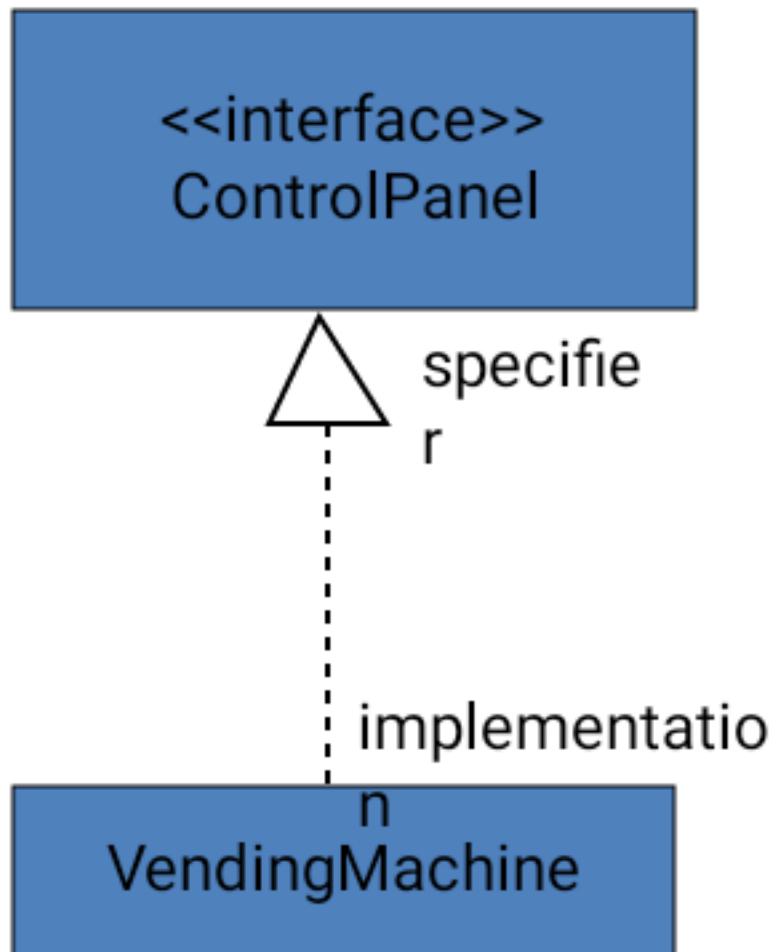
Interface Services

```
<<interface>>  
ControlPanel
```

```
getChoices : Choice[]  
makeChoice (c : Choice)  
getSelection : Selection
```

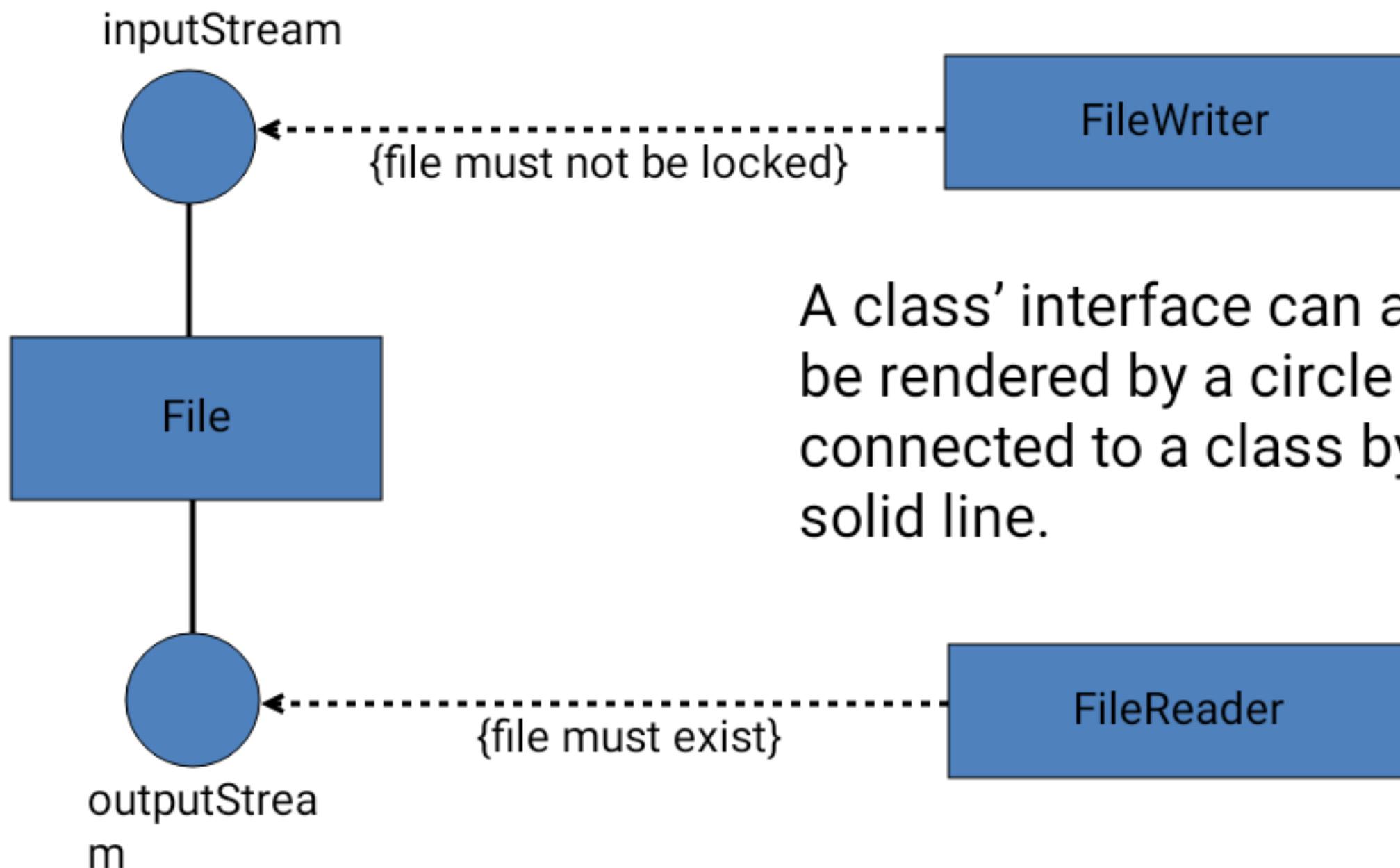
Interfaces do not get instantiated. They have no attributes or state. Rather, they specify the services offered by a related class.

Interface Realization Relationship



A *realization* relationship connects a class with an interface that supplies its behavioral specification. It is rendered by a dashed line with a hollow triangle towards the specifier.

Interfaces



A class' interface can also be rendered by a circle connected to a class by a solid line.

Class Diagrams

- Class Diagrams are like the paragraphs of a technical paper
 - each diagram should focus on a specific topic
 - a diagram provides supporting detail for the main concept(s) that it is trying to communicate
 - the level of the abstraction used in the diagrams should be consistent
- Together, all the diagrams for a system comprise a “model” of that system

Class Diagrams

- Pitfalls of Class Diagrams:
 - Using class diagrams alone can cause developers to **focus too much on structure and ignore behavior**
 - Using the wrong (or a mixed) perspective can lead to misunderstanding
 - Using the **wrong level of abstraction** can be confusing to the target audience
 - Using **mixed levels of abstraction** can reduce the usefulness of diagram

Bonus Slide!

- If you're interested in Auto-generating UML, Netbeans has an option to do it.
 - Install the UML plugin
 - Right-click on a project
 - Choose “Reverse Engineer”
 - Go to the new UML project
 - Select a package and choose to generate a new UML diagram

References

- ★ *Object-Oriented and Classical Software Engineering*, Sixth Edition, WCB/McGraw-Hill, 2005 Stephen R. Schach
- ★ UML resource page <http://www.uml.org/>

Component-Level Design

Slides adapted from various web sources with grateful acknowledgement of the many others who made their course materials freely available online.

Introduction

Background

- Component-level design occurs after the first iteration of the architectural design
- It strives to create a design model from the analysis and architectural models
 - The translation can open the door to subtle errors that are difficult to find and correct later
 - “Effective programmers should not waste their time debugging – they should not introduce bugs to start with.” **Edsger Dijkstra**
- A component-level design can be represented using some intermediate representation (e.g. graphical, tabular, or text-based) that can be translated into source code
- The design of data structures, interfaces, and algorithms should conform to well-established guidelines to help us avoid the introduction of errors

The Software Component

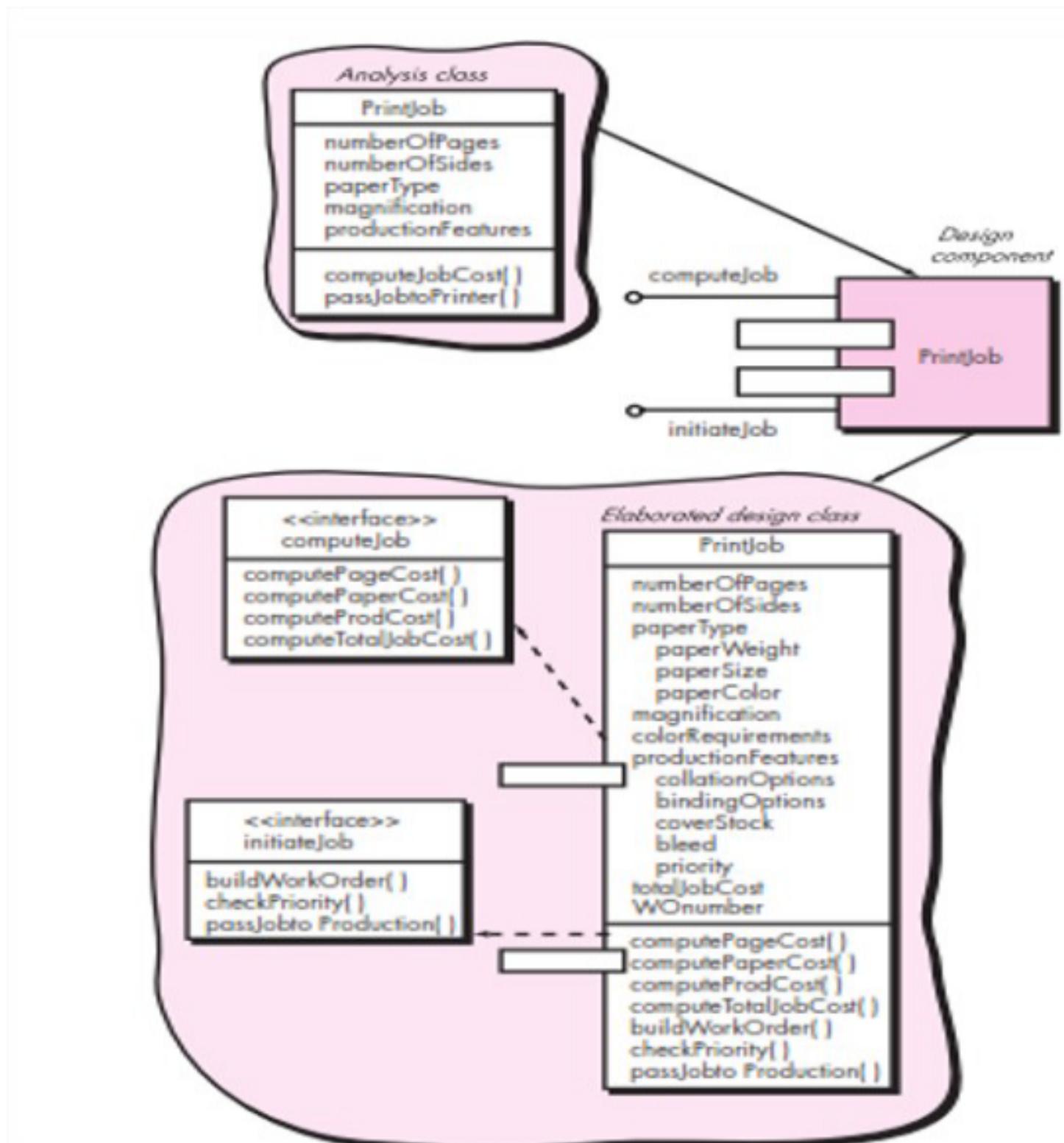
Defined

- A software component is a modular building block for computer software. Object Management Group (OMG) UML spec.
 - It is a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces
- A component communicates and collaborates with
 - Other components
 - Entities outside the boundaries of the system
- Three different views of a component
 - An object-oriented view
 - A conventional view
 - A process-related view

Object-oriented View

- A component is viewed as a set of one or more ~~collaborating classes~~
- Each problem domain (i.e., analysis) class and infrastructure (i.e., design) class is elaborated to identify all attributes and operations that apply to its implementation
 - This also involves defining the interfaces that enable classes to communicate and collaborate
- This elaboration activity is applied to every component defined as part of the architectural design
- Once this is completed, the following steps are performed
 - Provide further ~~elaboration~~ of each attribute, operation, and interface
 - Specify the ~~data structure~~ appropriate for each attribute
 - Design the ~~algorithmic detail~~ required to implement the processing logic associated with each operation
 - Design the mechanisms required to implement the ~~interface~~ to include the messaging that occurs between objects

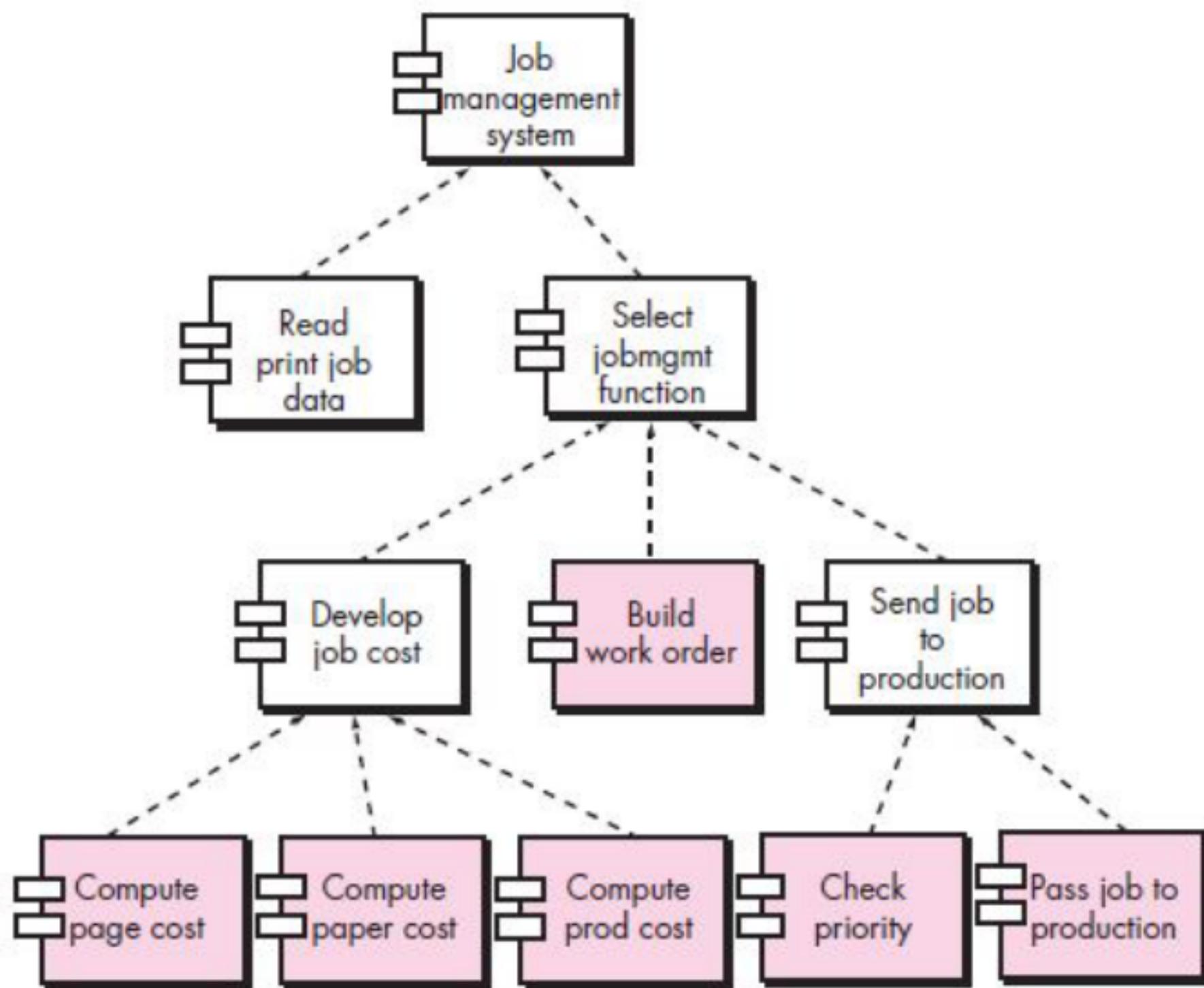
Elaboration of a design component



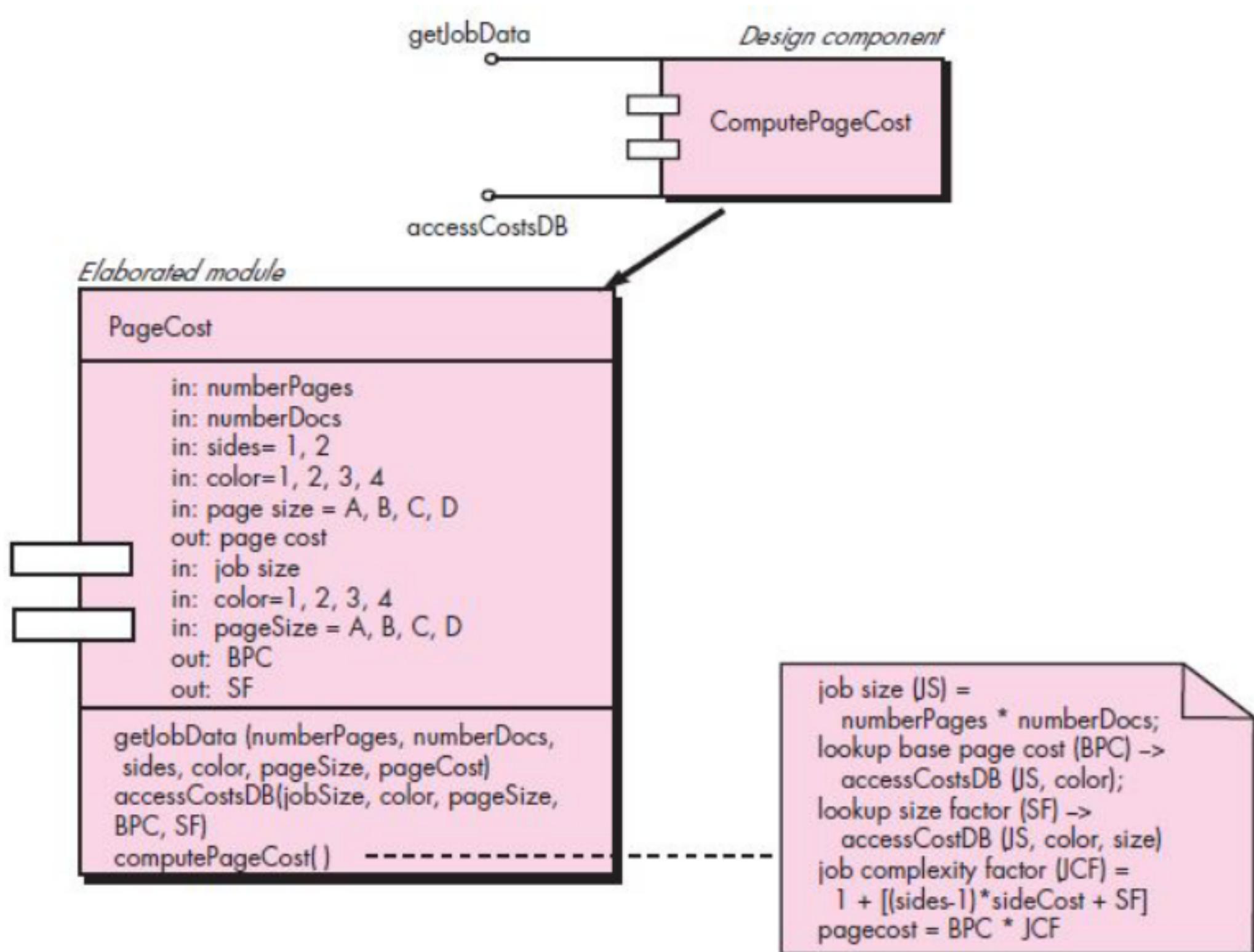
Conventional/Traditional View

- A component is viewed as a functional element (i.e., a module) of a program that incorporates
 - The processing logic
 - The internal data structures that are required to implement the processing logic
 - An interface that enables the component to be invoked and data to be passed to it
- A component serves one of the following roles
 - A control component that coordinates the invocation of all other problem domain components
 - A problem domain component that implements a complete or partial function that is required by the customer
 - An infrastructure component that is responsible for functions that support the processing required in the problem domain

Structure chart for a traditional system



Component-level design for *ComputePageCost*



Conventional View (continued)

- Conventional software components are derived from the data flow diagrams (DFDs) in the analysis model
 - Each transform bubble (i.e., module) represented at the lowest levels of the DFD is mapped into a module hierarchy
 - Control components reside near the top
 - Problem domain components and infrastructure components migrate toward the bottom
 - Functional independence is strived for between the transforms
- Once this is completed, the following steps are performed for each transform
 - Define the interface for the transform (the order, number and types of the parameters)
 - Define the data structures used internally by the transform
 - Design the algorithm used by the transform (using a stepwise refinement approach)

Process-related View

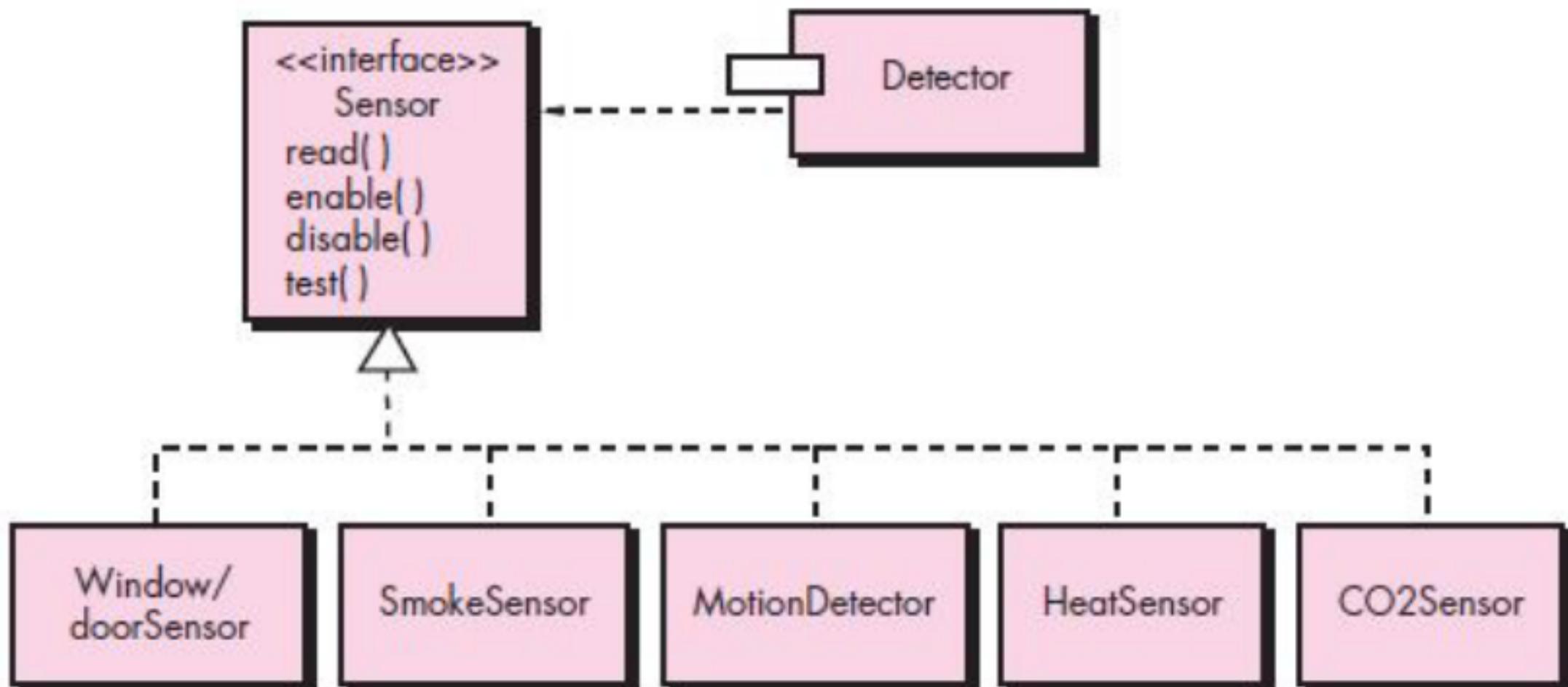
- Emphasis is placed on building systems from existing components maintained in a library rather than creating each component from scratch
- As the software architecture is formulated, components are selected from the library and used to populate the architecture
- Because the components in the library have been created with reuse in mind, each contains the following:
 - A complete description of their interface
 - The functions they perform
 - The communication and collaboration they require
- Component-based Software Engineering

Designing Class-Based Components

Component-level Design Principles

- Open-closed principle (OCP)

- A module or component should be open for extension but closed for modification
- The designer should specify the component in a way that allows it to be extended without the need to make internal code or design modifications to the existing parts of the component



Following the OCP

Component-level Design Principles

- **Liskov substitution principle (LSP)**
 - Subclasses should be substitutable for their base classes
 - A component that uses a base class should continue to function properly if a subclass of the base class is passed to the component instead
- **Dependency inversion principle (DIP)**
 - Depend on abstractions (i.e., interfaces); do not depend on concretions
 - The more a component depends on other concrete components (rather than on the interfaces) the more difficult it will be to extend
- **Interface segregation principle (ISP)**
 - Many client-specific interfaces are better than one general purpose interface
 - For a server class, specialized interfaces should be created to serve major categories of clients
 - Only those operations that are relevant to a particular category of clients should be specified in the interface

Component Packaging Principles

- Release reuse equivalency principle (REP)
 - The granularity of reuse is the granularity of release
 - Group the reusable classes into packages that can be managed, upgraded, and controlled as newer versions are created
- Common closure principle (CCP)
 - Classes that change together belong together
 - Classes should be packaged cohesively; they should address the same functional or behavioral area on the assumption that if one class experiences a change then they all will experience a change
- Common reuse principle (CRP)
 - Classes that aren't reused together should not be grouped together
 - Classes that are grouped together may go through unnecessary integration and testing when they have experienced no changes but when other classes in the package have been upgraded

Component-Level Design Guidelines

- Components
 - Establish naming conventions for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model
 - Obtain architectural component names from the problem domain and ensure that they have meaning to all stakeholders who view the architectural model (e.g., Calculator)
 - Use infrastructure component names that reflect their implementation-specific meaning (e.g., Stack)
- Dependencies and inheritance in UML
 - Model any dependencies from left to right and inheritance from top (base class) to bottom (derived classes)
 - Consider modeling any component dependencies as interfaces rather than representing them as a direct component-to-component dependency

Cohesion

- Cohesion is the “single-mindedness’ of a component
- It implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself
- The objective is to keep cohesion as **high** as possible
- The kinds of cohesion can be ranked in order from highest (best) to lowest (worst)
 - Functional
 - A module performs one and only one computation and then returns a result
 - Layer
 - A higher layer component accesses the services of a lower layer component
 - Communicational
 - All operations that access the same data are defined within one class

Cohesion (continued)

- Kinds of cohesion (continued)
 - Sequential
 - Components or operations are grouped in a manner that allows the first to provide input to the next and so on in order to implement a sequence of operations
 - Procedural
 - Components or operations are grouped in a manner that allows one to be invoked immediately after the preceding one was invoked, even when no data passed between them
 - Temporal
 - Operations are grouped to perform a specific behavior or establish a certain state such as program start-up or when an error is detected
 - Utility
 - Components, classes, or operations are grouped within the same category because of similar general functions but are otherwise unrelated to each other

Coupling

- As the amount of communication and collaboration increases between operations and classes, the complexity of the computer-based system also increases
- As complexity rises, the difficulty of implementing, testing, and maintaining software also increases
- Coupling is a qualitative measure of the degree to which operations and classes are connected to one another
- The objective is to keep coupling as low as possible

(More on next slide)

Coupling (continued)

- The kinds of coupling can be ranked in order from lowest (best) to highest (worst)
 - Data coupling
 - Operation A() passes one or more atomic data operands to operation B(); the less the number of operands, the lower the level of coupling
 - Stamp coupling
 - A whole data structure or class instantiation is passed as a parameter to an operation
 - Control coupling
 - Operation A() invokes operation B() and passes a control flag to B that directs logical flow within B()
 - Consequently, a change in B() can require a change to be made to the meaning of the control flag passed by A(), otherwise an error may result
 - Common coupling
 - A number of components all make use of a global variable, which can lead to uncontrolled error propagation and unforeseen side effects
 - Content coupling
 - One component secretly modifies data that is stored internally in another component

(More on next slide)

Coupling (continued)

- Other kinds of coupling (unranked)
 - Subroutine call coupling
 - When one operation is invoked it invokes another operation within side of it
 - Type use coupling
 - Component A uses a data type defined in component B, such as for an instance variable or a local variable declaration
 - If/when the type definition changes, every component that declares a variable of that data type must also change
 - Inclusion or import coupling
 - Component A imports or includes the contents of component B
 - External coupling
 - A component communicates or collaborates with infrastructure components that are entities external to the software (e.g., operating system functions, database functions, networking functions)

Conducting Component-Level Design

- Identify all design classes that correspond to the problem domain as defined in the analysis model and architectural model
- Identify all design classes that correspond to the infrastructure domain
 - These classes are usually not present in the analysis or architectural models
 - These classes include GUI components, operating system components, data management components, networking components, etc.
- Elaborate all design classes that are not acquired as reusable components
 - Specify message details (i.e., structure) when classes or components collaborate
 - Identify appropriate interfaces (e.g., abstract classes) for each component
 - Elaborate attributes and define data types and data structures required to implement them (usually in the planned implementation language)
 - Describe processing flow within each operation in detail by means of pseudocode or UML activity diagrams

(More on next slide)

Conducting Component-Level Design (continued)

- Describe persistent data sources (databases and files) and identify the classes required to manage them
- Develop and elaborate behavioral representations for a class or component
 - This can be done by elaborating the UML state diagrams created for the analysis model and by examining all use cases that are relevant to the design class
- Elaborate deployment diagrams to provide additional implementation detail
 - Illustrate the location of key packages or classes of components in a system by using class instances and designating specific hardware and operating system environments
- Factor every component-level design representation and always consider alternatives
 - Experienced designers consider all (or most) of the alternative design solutions before settling on the final design model
 - The final decision can be made by using established design principles and guidelines

Designing Conventional/ Traditional Components

Introduction

- Proposed by Edsger Dijkstra in late 1960s.
- Conventional design constructs emphasize the maintainability of a functional/procedural program
 - Sequence, condition, and repetition
- Each construct has a predictable logical structure where control enters at the top and exits at the bottom, enabling a maintainer to easily follow the procedural flow
- Use of limited number of logical constructs also contributes to a human understanding process that psychologists call “**chunking**”

Introduction

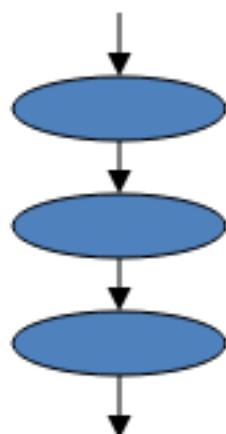
- Various notations depict the use of these constructs
 - Graphical design notation (Control Flow Graphs - CFG)
 - Sequence, if-then-else, selection, repetition (see next slide)
 - Tabular design notation (see upcoming slide)
 - Program Design Language (PDL)
 - Similar to a programming language; however, it uses narrative text embedded directly within the program statements

Control Flow Graphs

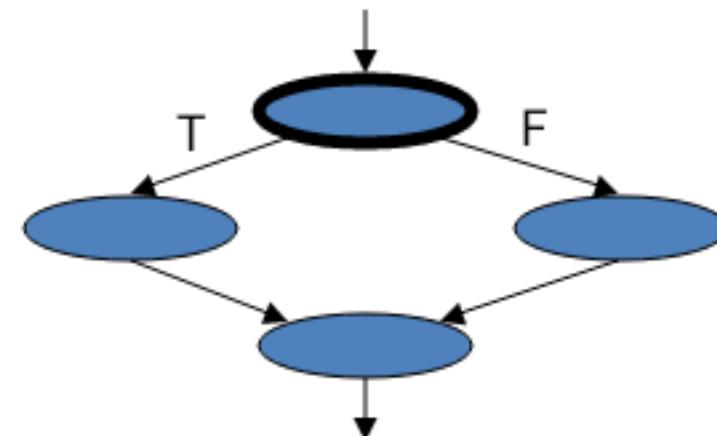
- A Control Flow Graph is a representation of the different blocks of code in a program, and the different paths that the interpreter/compiler can take through the code.
- originally developed by *Frances E. Allen* – American Computer Scientist (*In 2006 became the first woman to win the Turing Award*)
- Used in static analysis as well as compiler applications, as they can accurately represent the flow inside of a program unit
- **Characteristics:**
 - shows all the paths that can be traversed during a program execution
 - directed graph
 - Edges in CFG portray control flow paths and the nodes in

CFG portray basic blocks.

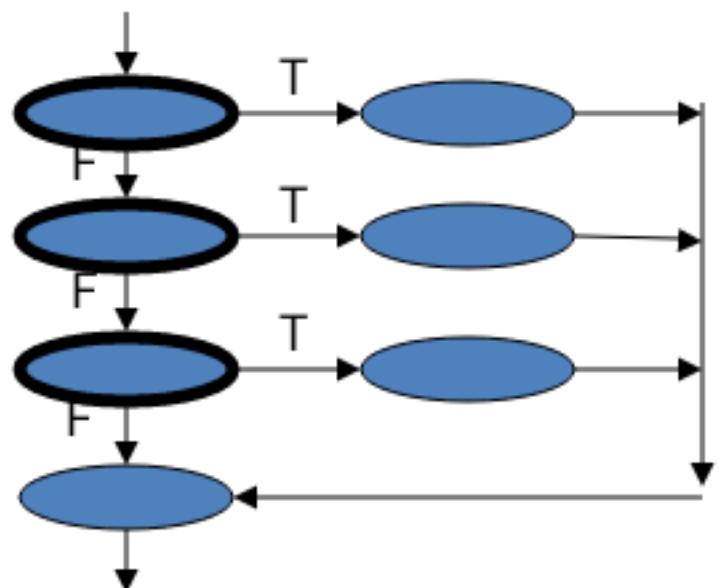
Notations for CFG



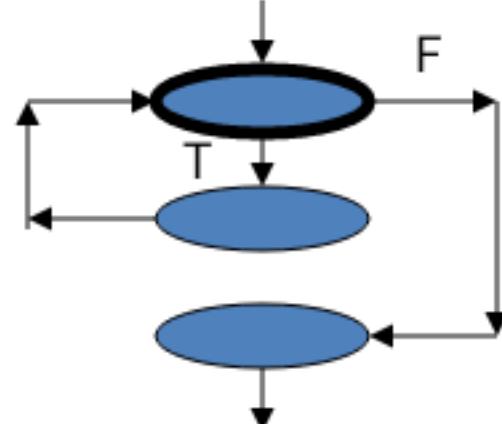
Sequence



If-then-else

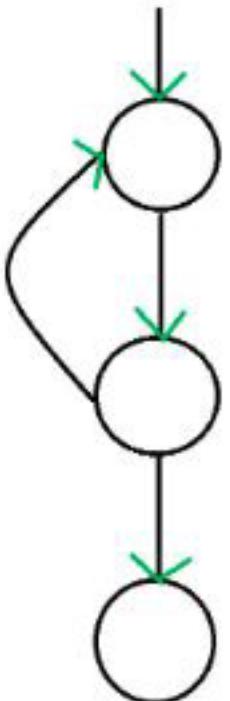


Selection

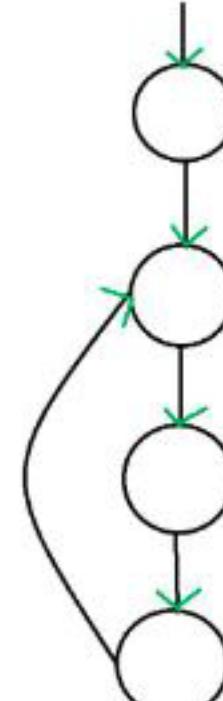


Repetition (While loop)

Notations for CFG contd'



do-while



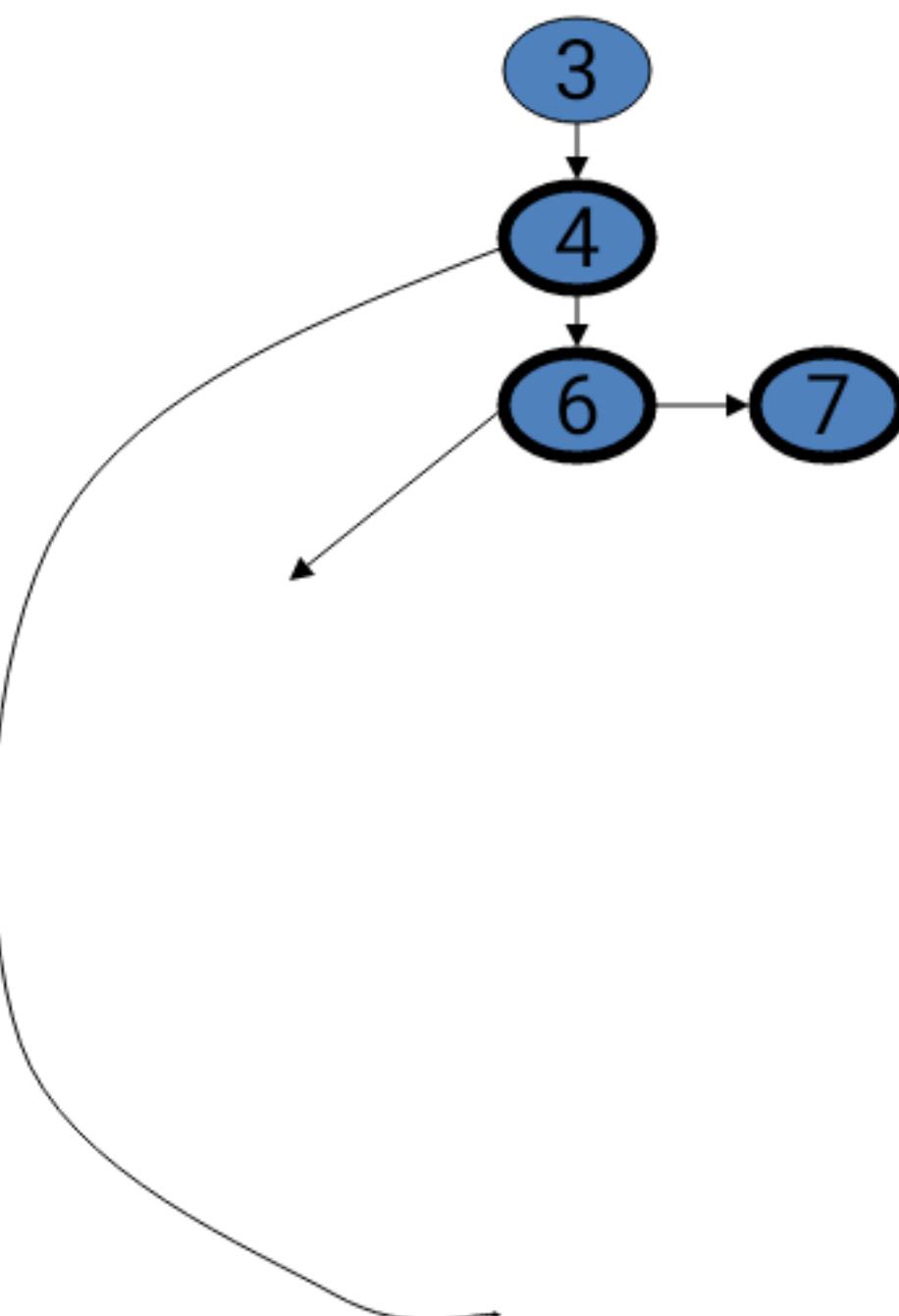
for

CFG – Example

```
1 int functionZ(int y)
2 {
3     int x = 0;
4
5     while (x <= (y * y))
6     {
7         if ((x % 11 == 0) &&
8             (x % y == 0))
9         {
10            printf("%d", x);
11            x++;
12        } // End if
13        else if ((x % 7 == 0) ||
14                  (x % y == 1))
15        {
16            printf("%d", y);
17            x = x + 2;
18        } // End else
19    } // End while
20
21    printf("\n");
22} // End functionZ
```

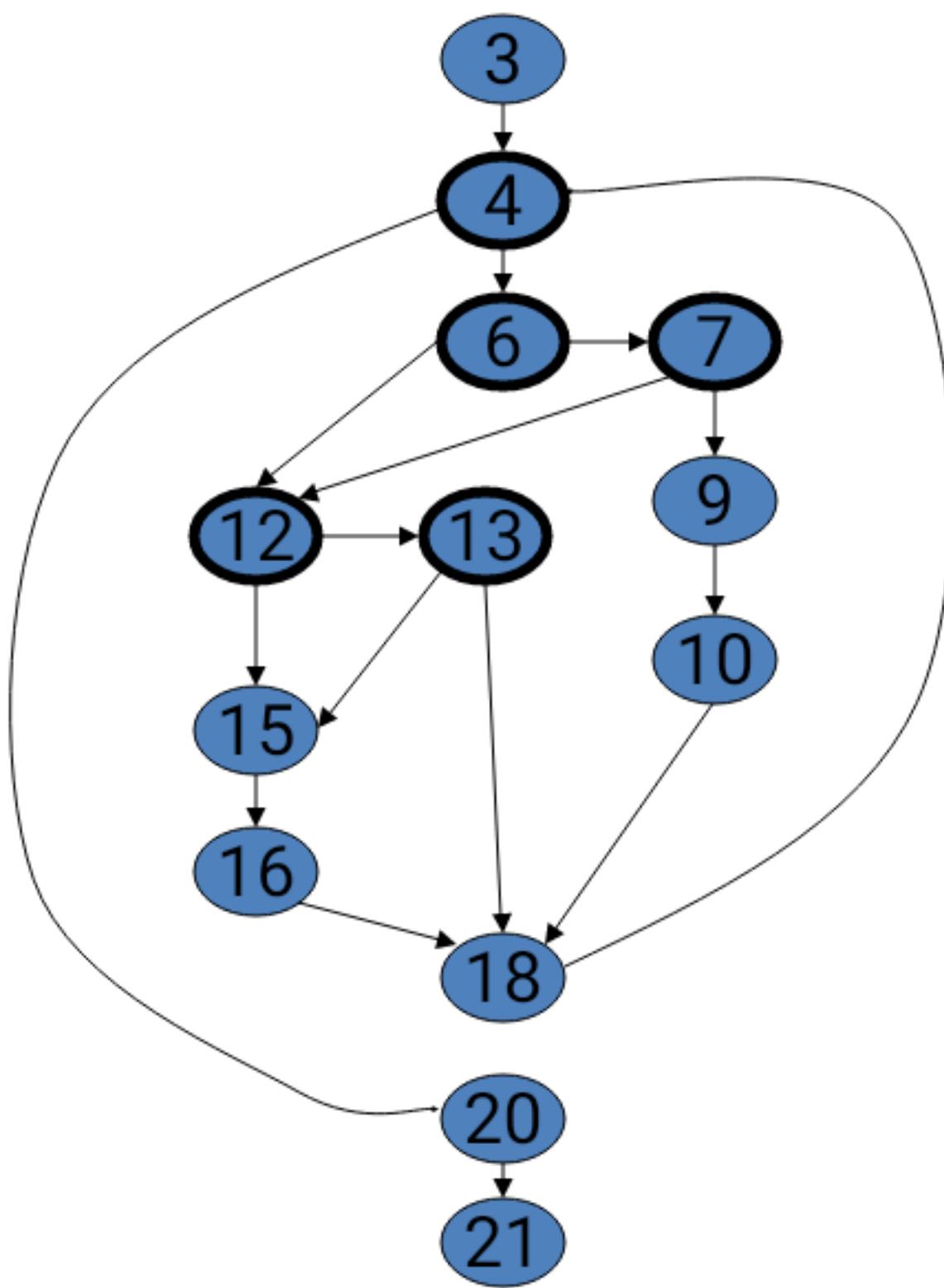
CFG – Example contd'

```
1 int functionZ(int y)
2 {
3     int x = 0;
4
5     while (x <= (y * y))
6     {
7         if ((x % 11 == 0) &&
8             (x % y == 0))
9         {
10            printf("%d", x);
11            x++;
12        } // End if
13        else if ((x % 7 == 0) ||
14                  (x % y == 1))
15        {
16            printf("%d", y);
17            x = x + 2;
18        } // End else
19    } // End while
20
21    printf("\n");
22} // End functionZ
```



CFG – Example contd'

```
1 int functionZ(int y)
2 {
3     int x = 0;
4
5     while (x <= (y * y))
6     {
7         if ((x % 11 == 0) &&
8             (x % y == 0))
9         {
10            printf("%d", x);
11            x++;
12        } // End if
13        else if ((x % 7 == 0) ||
14                  (x % y == 1))
15        {
16            printf("%d", y);
17            x = x + 2;
18        } // End else
19    } // End while
20
21    printf("\n");
22 } // End functionZ
```



Tabular Design Notation

- List all actions that can be associated with a specific procedure (or module)
- List all conditions (or decisions made) during execution of the procedure
- Associate specific sets of conditions with specific actions, eliminating impossible combinations of conditions; alternatively, develop every possible permutation of conditions
- Define rules by indicating what action(s) occurs for a set of conditions

Program Design Language

Program design language (PDL), also called *structured English* or *pseudocode*, incorporates the logical structure of a programming language with the free-form expressive ability of a natural language (e.g., English).

Refer text book for an example of *SafeHome* Security function.

(More on next slide)

Tabular Design Notation

	Rules					
Conditions	1	2	3	4	5	6
Regular customer	T	T				
Silver customer			T	T		
Gold customer					T	T
Special discount	F	T	F	T	F	T
Actions						
No discount		✓				
Apply 8 percent discount			✓	✓		
Apply 15 percent discount					✓	✓
Apply additional x percent discount	✓		✓		✓	

Entity Relationship Diagram (ERD)

Slides adapted from various web sources with grateful acknowledgement of the many others who made their course materials freely available online.

As part of the Requirements Elaboration:

DFD and ERD

Basic Concepts and Notations on Entity Relationship Diagram

What is an ERD?

- An Entity Relationship Diagram (ERD) is a graphical representation of an organization's data storage requirements
- ERDs are abstractions of the real world, which simplify the problem to be solved while retaining its essential features

Why ERD?

- **Entity relationship diagrams are used to**
 - Identify the data that must be captured, stored and retrieved in order to support the activities performed by an organization
 - A major data modeling tool and help to organize the data in the project into entities and define the relationship between entities
 - It enables the analyst to produce a good database structure so that the data can be stored and retrieved in a most efficient manner

Composition of ERD

- Entity relationship diagrams have three components
 - Entities
 - Attributes
 - Relationships

Entity in ERD

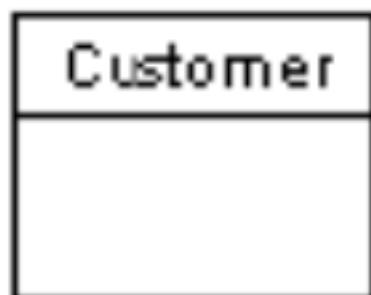
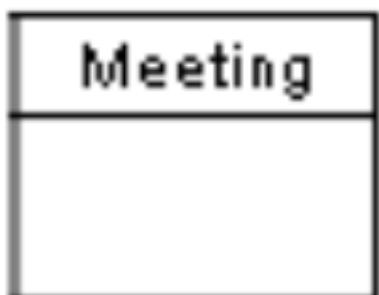
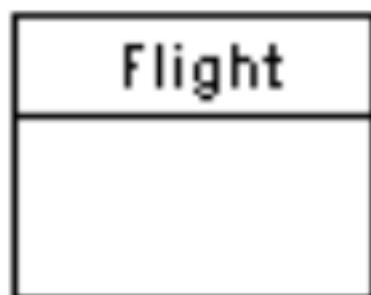
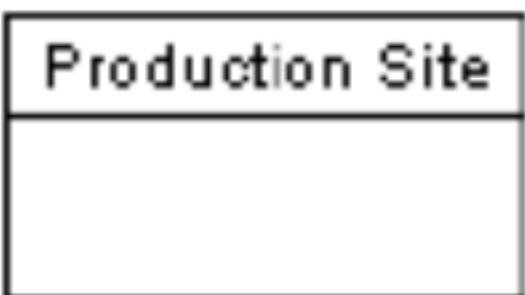
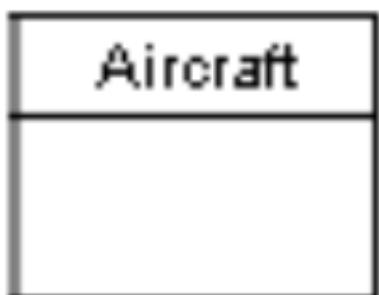
● Entities

- An entity is anything real or abstract about which we want to store data. In short, anything, which an organization needs to store data about
- Entity types fall into five classes
 - Roles
 - e.g. *Employee, Student*
 - Events
 - e.g. *Payment, Borrow*
 - Locations
 - e.g. *Campus, City*
 - Tangible things or concepts
 - e.g. *Department, Book*

Representation Entity

- Entities

- Entities are represented on the diagram by labeled boxes



Entity Sub-type

- **Entities**

- Sometimes it is useful to generalize a group of entities

Example

CAR, SHIP and AEROPLANE all are type of VEHICLE

Attributes in ERD

● Attributes

- Entities are further described by their attributes (also called data elements)
- These are the smallest units of data that can be described in a meaningful manner

Example: An EMPLOYEE entity may have the following attributes

Employee
Employee Number
Surname
Given Name
Date of Birth
Telephone Number
Department

Types of Attributes (1)

- Simple
 - Each entity has a single atomic value for the attribute. For example, SSN or Sex.
- Composite
 - The attribute may be composed of several components. For example:
 - Address(Apt#, House#, Street, City, State, ZipCode, Country), or
 - Name(FirstName, MiddleName, LastName).
 - Composition may form a hierarchy where some components are themselves composite.
- Multi-valued
 - An entity may have multiple values for that attribute. For example, Color of a CAR or PreviousDegrees of a STUDENT.
 - Denoted as {Color} or {PreviousDegrees}.

Types of Attributes (2)

- In general, composite and multi-valued attributes may be nested arbitrarily to any number of levels, although this is rare.
 - For example, PreviousDegrees of a STUDENT is a composite multi-valued attribute denoted by {PreviousDegrees (College, Year, Degree, Field)}
 - Multiple PreviousDegrees values can exist
 - Each has four subcomponent attributes:
 - College, Year, Degree, Field

Relationships in ERD

- **Relationships**

- A data relationship is a natural association that exist between one or more entities

Example

EMPLOYEE works in DEPARTMENT

EQUIPMENT is allocated to PROJECT

ESTEEM is a type of CAR

One-to-one Relationships

- Three types of relationship exist between two different entities
- It is also called the **cardinality** (the number of occurrences of one entity for a single occurrence of the related entity)
 - One-to-one relationship
 - One-to-many relationship
 - Many-to-many relationship

Example of a composite attribute

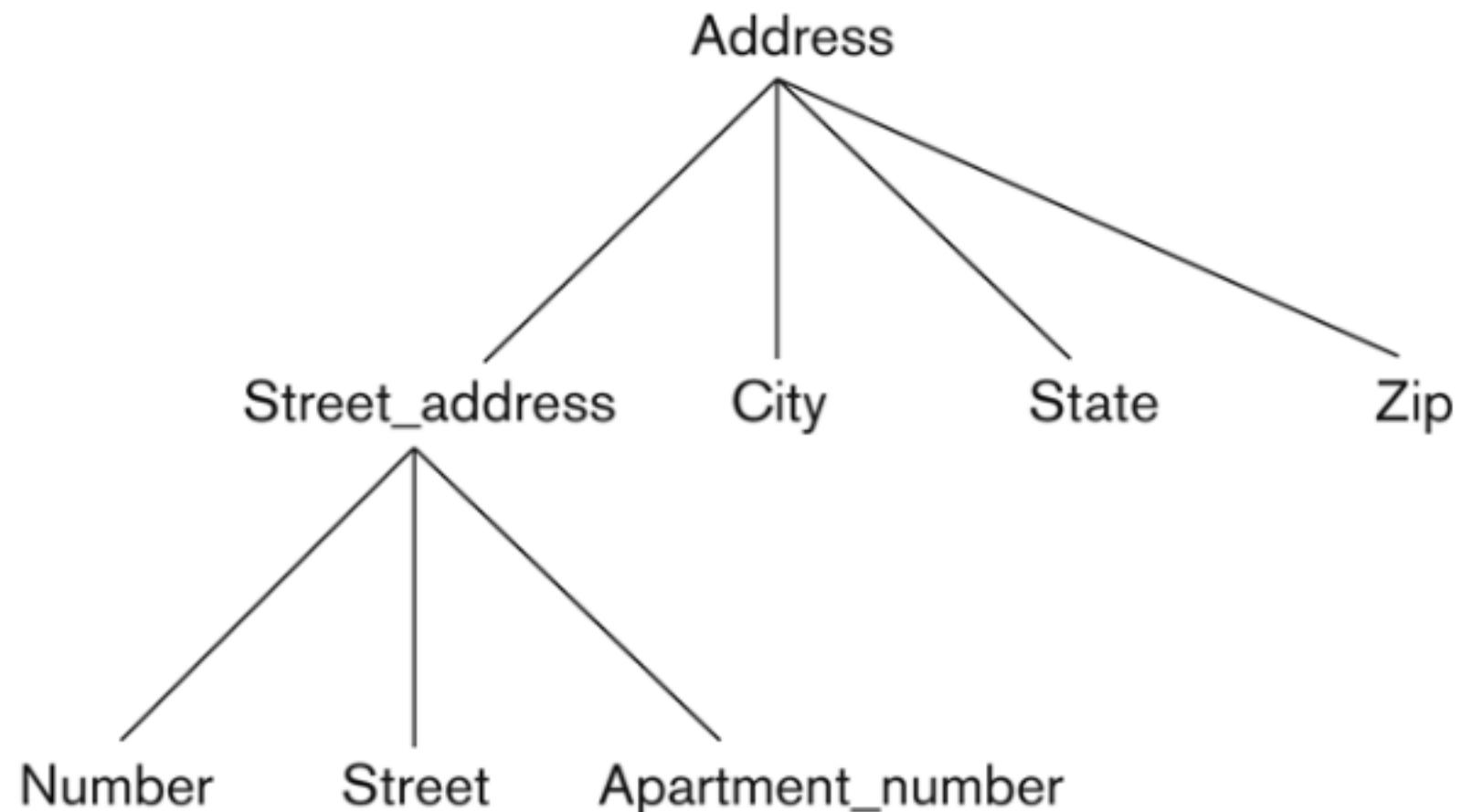


Figure 3.4
A hierarchy of
composite attributes.

One-to-one Relationships

- This type of relationship takes place when a single occurrence of an entity is related to just one occurrence of a second entity

Example:

PERSON **owns** a CAR

CAR is owned by a PERSON

One-to-many Relationships

- This type of relationship takes place when a single occurrence of an entity is related to many occurrence of a second entity

Example:

DEPARTMENT **has** EMPLOYEE

EMPLOYEE **works in** DEPARTMENT

Many-to-many Relationships

- This type of relationship takes place when many occurrence of an entity are related to many occurrence of a second entity

Example:

STAFF work in PROJECT

PROJECT assigned to STAFF

Recursive Relationships

- Relationship can exist between different occurrences of the same type of entity

Example:

EMPLOYEE manager EMPLOYEE

NETWORK makes NETWORK

Entity Relationship Diagram Methodology

ERD Methodology

10 basic steps

- Identify entities
- Find relationship
- Draw rough ERD
- Fill in cardinality
- Define Primary keys(s)
- Draw key-based ERD
- Identify attributes
- Map attributes
- Draw fully attributes
- Check results

A Simple Example

A company has several departments.
Each department has a supervisor and at least one employee. Employees must be assigned to at least one, but possibly more departments.
At least one employee is assigned to a project, but an employee may be on vacation and not assigned to any projects.
The important data fields are the names of the departments, projects, supervisors, and employees, as well as, the supervisor numbers, employee numbers and project numbers.

1. Identify Entities

Objective: Identify the roles, events, locations, tangible things or concepts about which the end-user want to store data

A company has several departments.

Each **department** has a **supervisor** and at least one **employee**. Employees must be assigned to at least one, but possibly more departments.

At least one employee is assigned to a **project**, but an employee may be on vacation and not assigned to any projects.

The important data fields are the names of the departments, projects, supervisors, and employees, as well as, the supervisor numbers, employee numbers and project numbers.

Note: One tempted to make **company** an entity, but it is a false entity because it has only one instance in this problem. True entities must have more than one instance.

2. Find Relationships

Objective: Find the natural association between pair of entities using a
relation
ship matrix

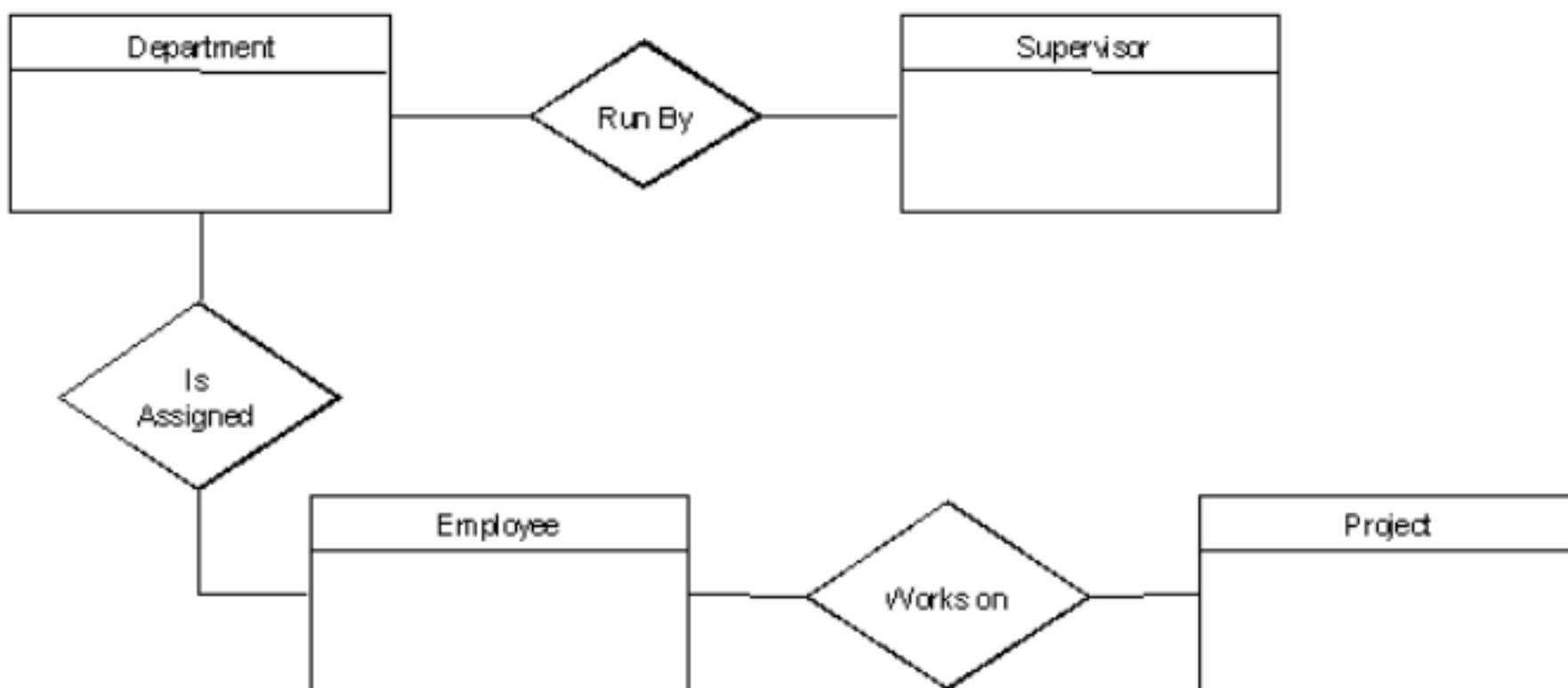
A company has several departments.

Each **department** has a **supervisor** and at least one **employee**. **Employees** must be assigned to at least one, but possibly more **departments**.

At least one **employee** is assigned to a **project**, but an **employee** may be on vacation and not assigned to any **projects**.

3. Draw Rough ERD

Objective: Put entities in rectangles and relationships on line segments connecting the entities



4. Fill in Cardinality

Objective: Determine the number of occurrences of one entity for a single occurrence of the related entity

A company has several departments.

Each department has a supervisor and at least one employee.

Employees must be assigned to at least one, but possibly more departments.

At least one employee is assigned to a project, but an employee may be on vacation and not assigned to any projects.

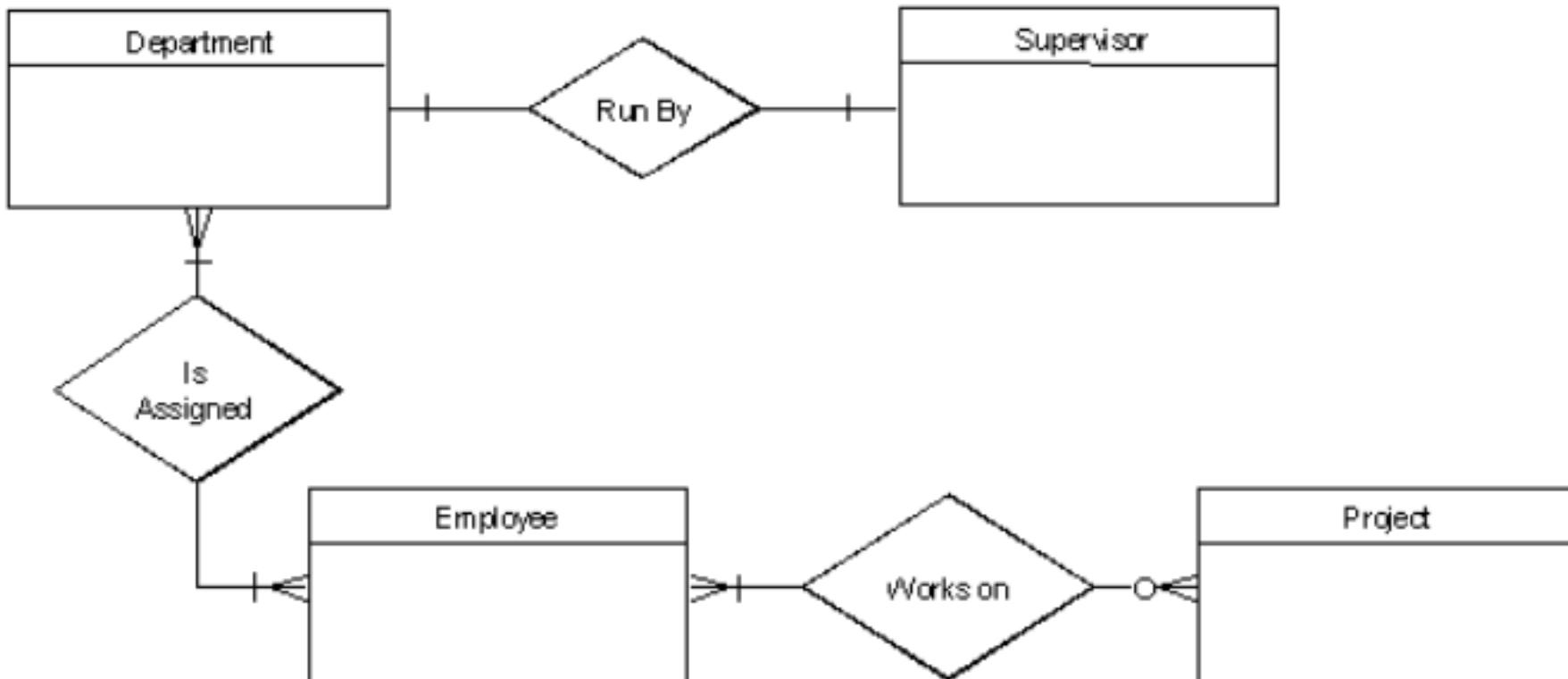
From the description of the problem:

- Each Department has exactly one Supervisor
- A Supervisor is in charge of one and only one Department
- Each Department is assigned to at least one Employee
- Each Employee works for at least one Department
- Each Project has at least one Employee working on it
- An Employee is assigned to 0 or more project

4. Fill in Cardinality

From the description of the problem:

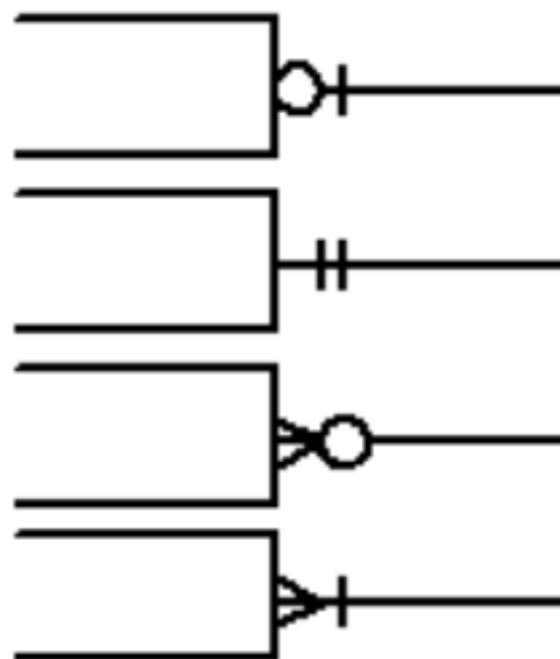
- Each Department **has exactly one** Supervisor
- A Supervisor is in charge of **one and only one** Department
- Each Department is assigned to at least **one** Employee
- Each Employee works for **at least one** Department
- Each Project has **at least one** Employee working on it
- An Employee is assigned to **0 or more** project



Crow's Foot Notation (Gordon Everest'1976)



Summary of Crow's Foot Notation



One or Zero

One and only One

Zero or Many

One or Many

5. Define Primary Keys

A company has several departments.

Each department has a supervisor and at least one employee.

Employees must be assigned to at least one, but possibly more departments.

At least one employee is assigned to a project, but an employee may be on vacation and not assigned to any projects.

The important data fields are the [names of the departments](#), projects, supervisors, and employees, as well as, the [supervisor numbers](#), [employee numbers](#) and [project numbers](#).

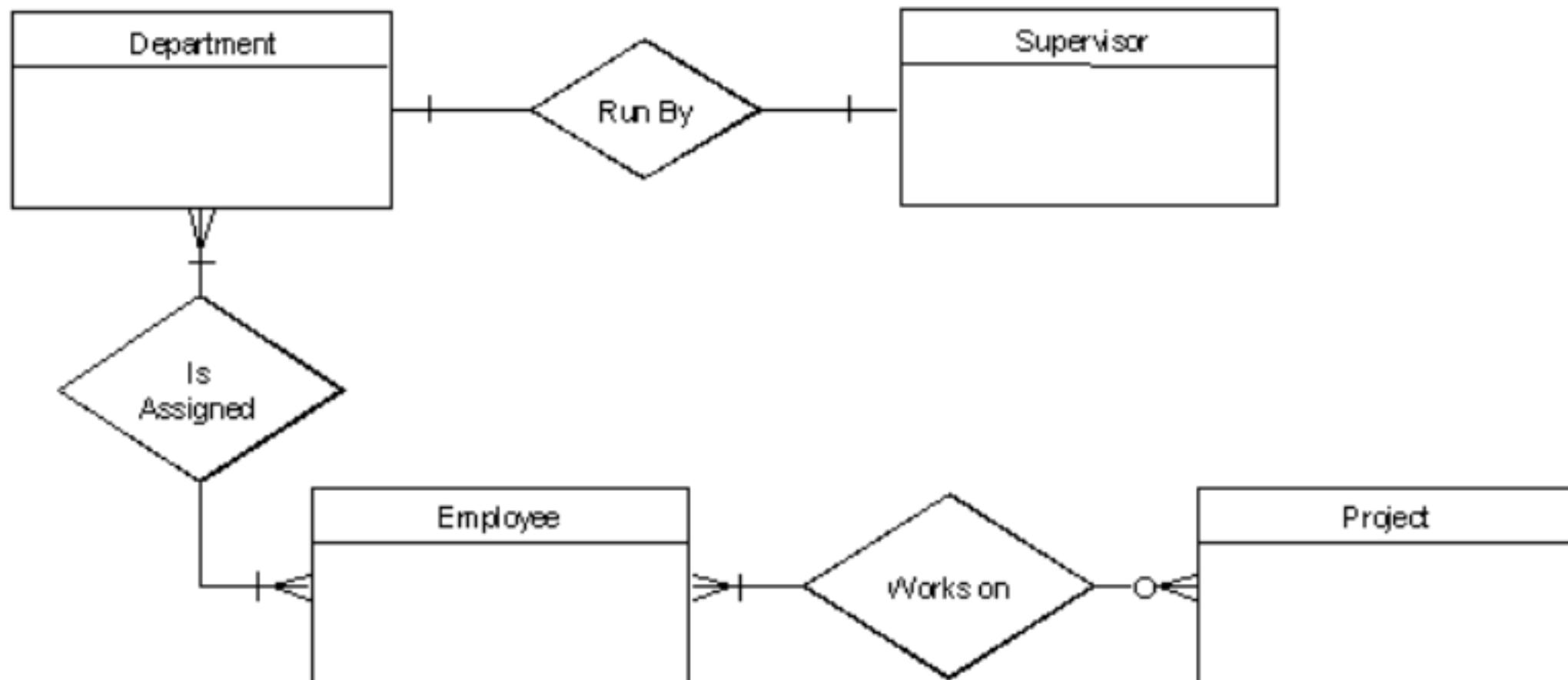
An attribute of an entity type for which each entity must have a unique value is called a primary key attribute of the entity type.

The Primary keys are

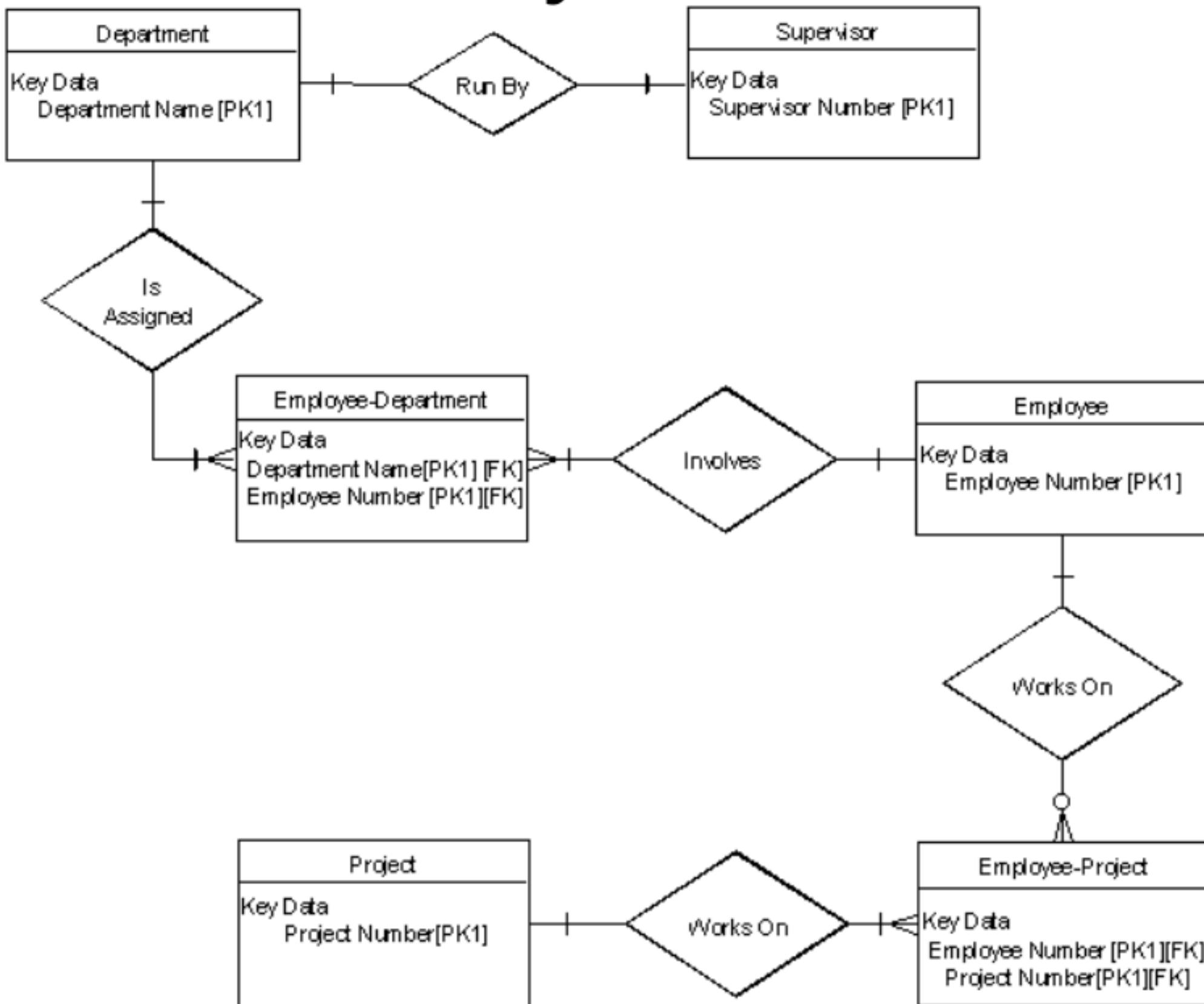
- Department name
- Supervisor number
- Employee number
- Project number

6. Draw Key-based ERD

Objective: Eliminate many-to-many relationship and include primary and foreign key in each entity



6. Draw Key-based ERD



7. Identify Attributes

Objective: Name the information details (fields), which are essential to the system under development

The only attributes indicated are the names of

A company has several departments.

Each department has a supervisor and at least one employee. Employees must be assigned to at least one, but possibly more departments.

At least one employee is assigned to a project, but an employee may be on vacation and not assigned to any projects.

The important data fields are the **names of the departments, projects, supervisors, and employees**, as well as, the **supervisor numbers, employee numbers** and **project numbers**.

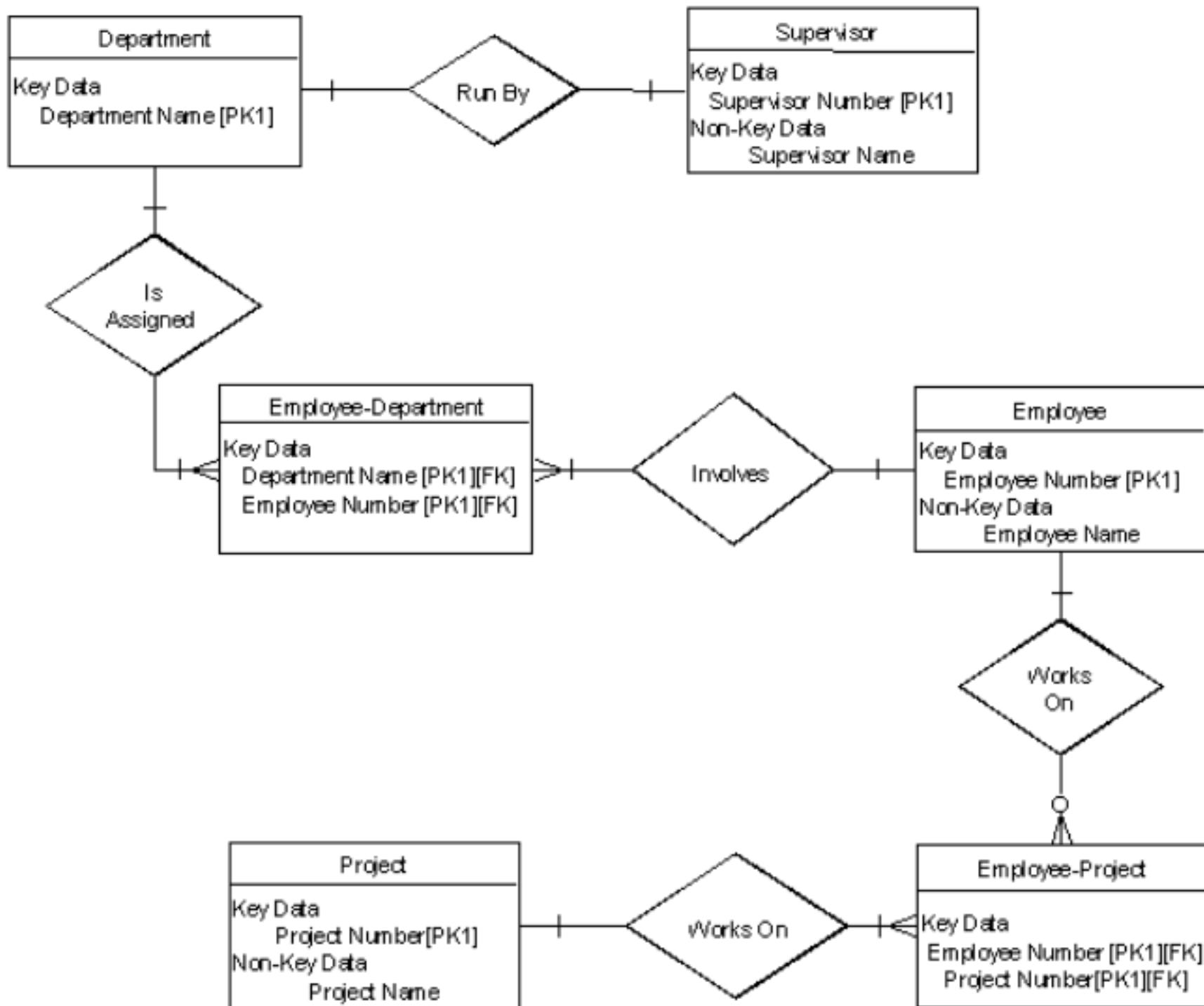
8. Map Attributes

Objective: For each attribute, match it with exactly one entity that it describes

9. Draw Fully Attributed ERD

Objective: Adjust the ERD from Step 6 to account for entities or relationships discovered in Step 8.

9. Draw Fully Attributed ERD



10. Draw Fully Attributed ERD

Objective: Does the final ERD accurately depict the system data?

The Final ERD appears to model the data in this system well.

User Interface Analysis and Design

- Introduction
- Golden rules of user interface design
- Reconciling four different models
- User interface analysis
- User interface design
- User interface evaluation
- Example user interfaces

Introduction

Interface Design

Easy to learn?

Easy to use?

Easy to understand?



Interface Design

Typical Design Errors

- lack of consistency**
- too much memorization**
- no guidance / help**
- no context sensitivity**
- poor response**
- Arcane/unfriendly**



UX, UI and Usability

The difference

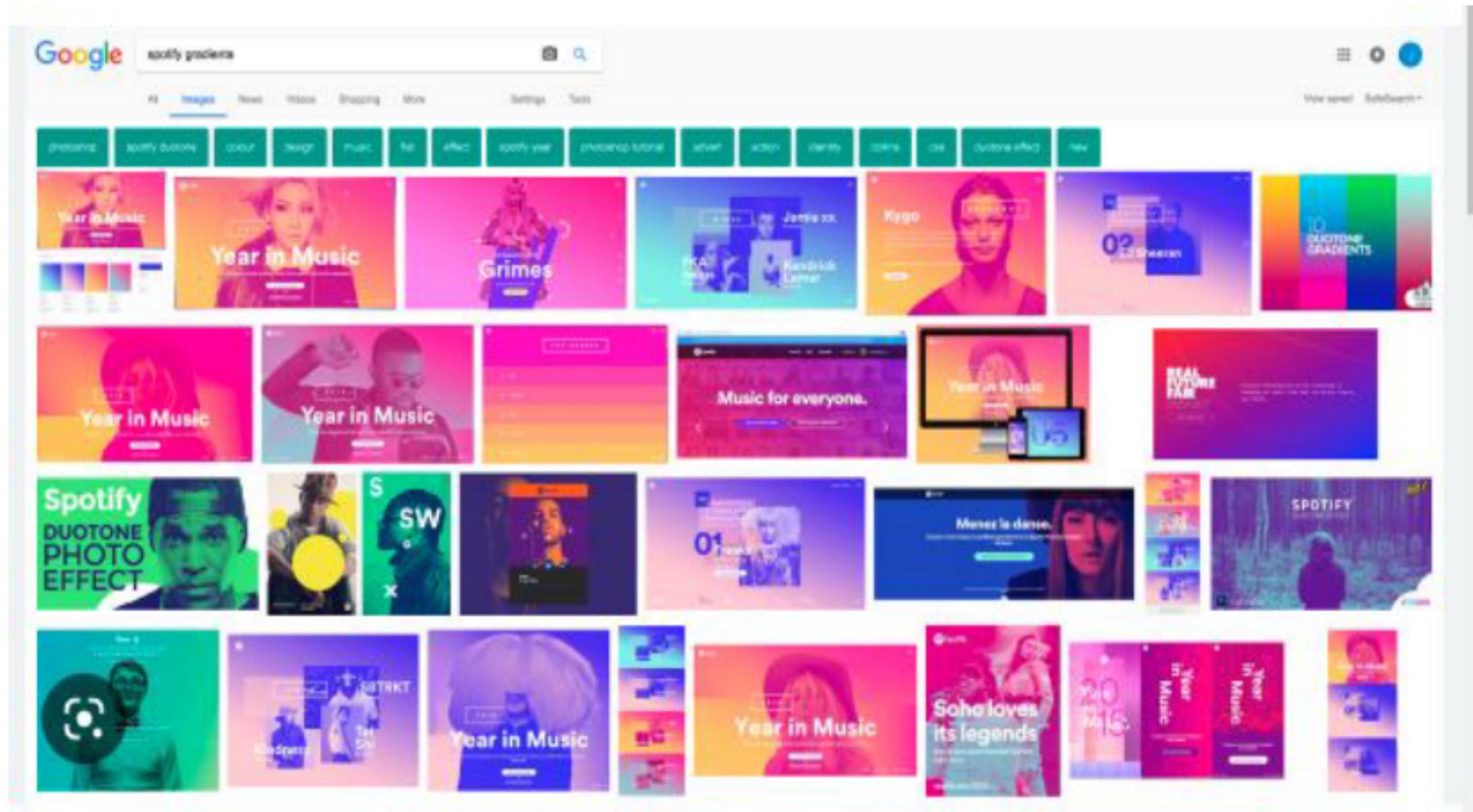
www.useit.com

- Limited short-term memory
 - People can instantaneously remember about 7 items of information. If you present more than this, they are more liable to make mistakes.
- People make mistakes
 - When people make mistakes and systems go wrong, inappropriate alarms and messages can increase stress and hence the likelihood of more mistakes.
- People are different
 - People have a wide range of physical capabilities. Designers should not just design for their own capabilities.
- People have different interaction preferences
 - Some like pictures, some like text.

Human factors in interface design

- Limited short-term memory
 - People can instantaneously remember about 7 items of information. If you present more than this, they are more liable to make mistakes.
- People make mistakes
 - When people make mistakes and systems go wrong, inappropriate alarms and messages can increase stress and hence the likelihood of more mistakes.
- People are different
 - People have a wide range of physical capabilities. Designers should not just design for their own capabilities.
- People have different interaction preferences
 - Some like pictures, some like text.

Multiple user interfaces



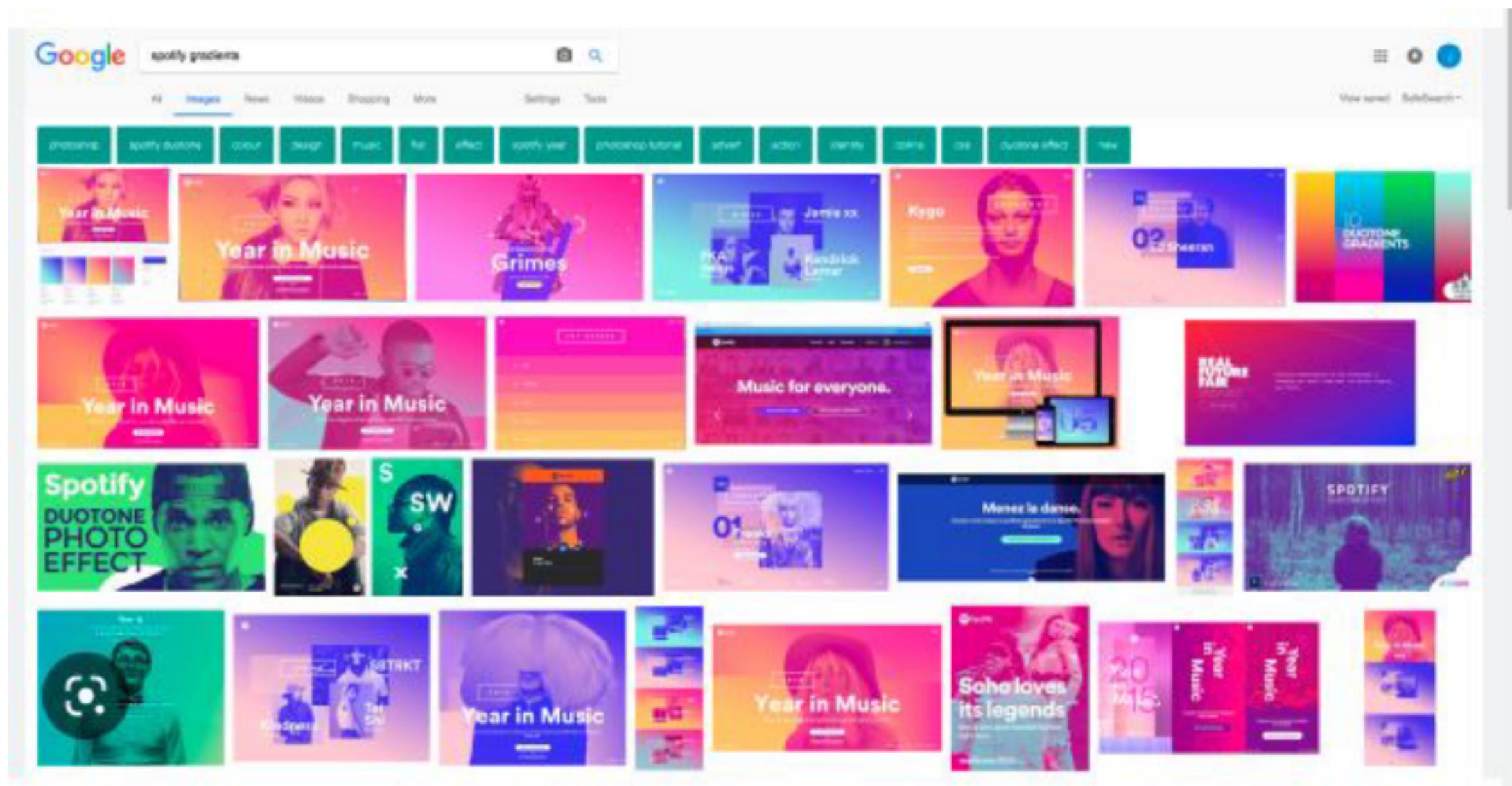
Background

- Interface design focuses on the following
 - The design of interfaces between software components
 - The design of interfaces between the software and other nonhuman producers and consumers of information
 - The design of the interface between a human and the computer
- Graphical user interfaces (GUIs) have helped to eliminate many of the most horrific interface problems
- However, some are still difficult to learn, hard to use, confusing, counterintuitive, unforgiving, and frustrating
- User interface analysis and design has to do with the study of people and how they relate to technology

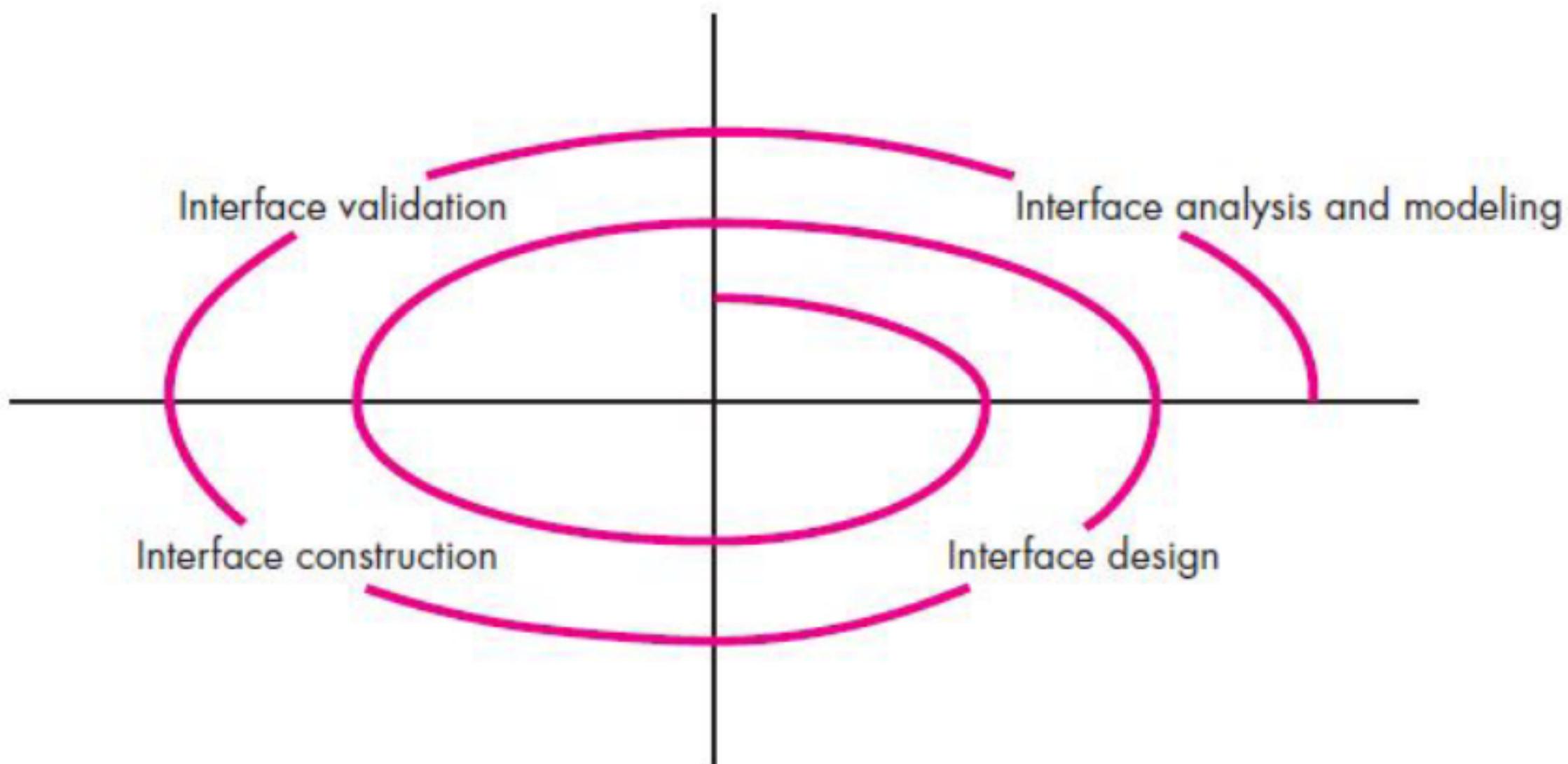
9 of The Best UI Design Examples in 2023

- Dribbble's card design
- Mailchimp's usability
- Dropbox's responsive color system
- Pinterest's waterfall effect
- Hello Monday's white space
- Current app's color palette
- Rally's dynamism
- Cognito's custom animation

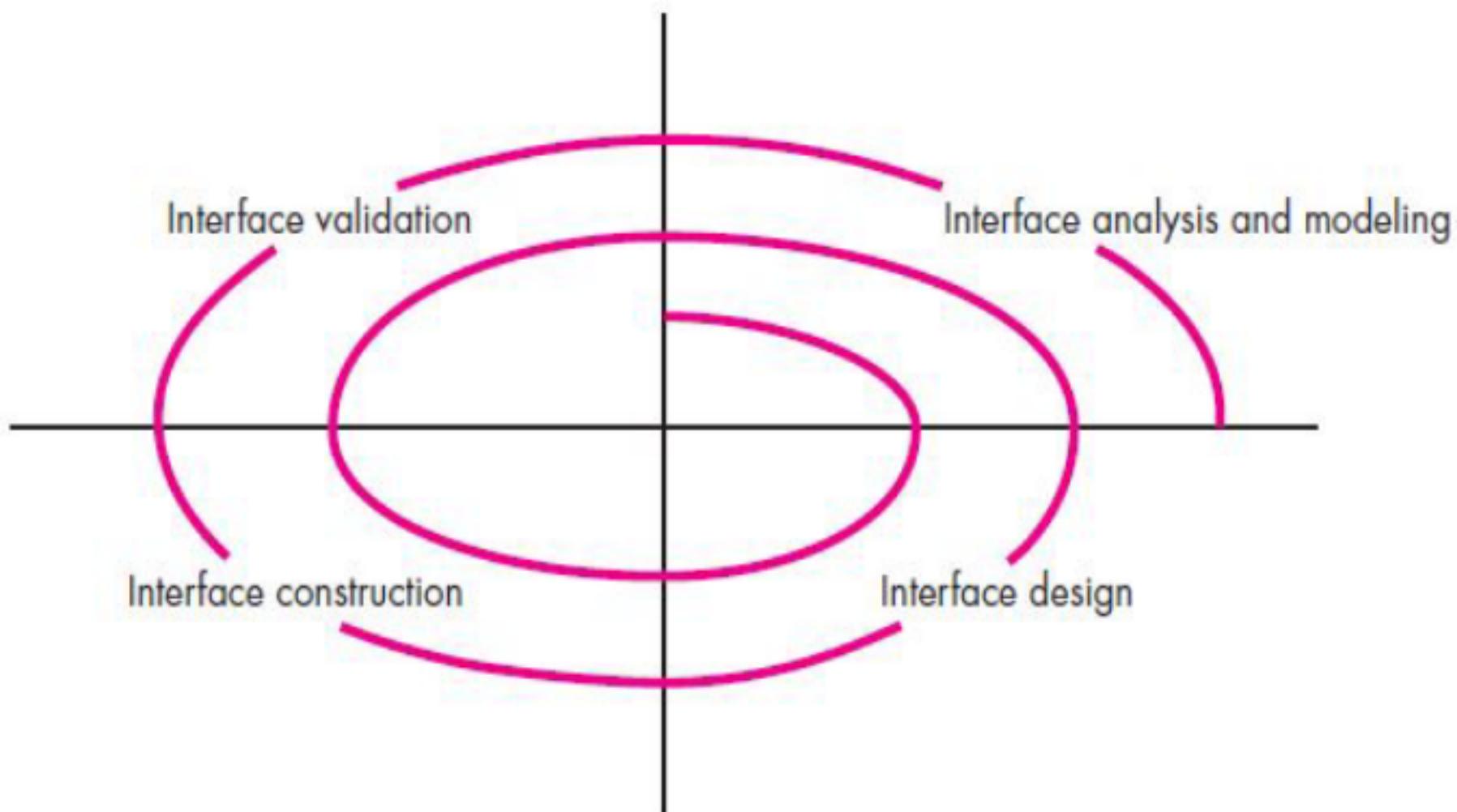
Spotify's color gradient



Spotify's color gradient



The user interface design process



A Spiral Process

- User interface development follows a spiral process
 - **Interface analysis (user, task, and environment analysis)**
 - Focuses on the profile of the users who will interact with the system
 - Concentrates on users, tasks, content and work environment
 - Studies different models of system function (as perceived from the outside)
 - Delineates the human- and computer-oriented tasks that are required to achieve system function
 - **Interface design**
 - Defines a set of interface objects and actions (and their screen representations) that enable a user to perform all defined tasks in a manner that meets every usability goal defined for the system
 - **Interface construction**
 - Begins with a prototype that enables usage scenarios to be evaluated
 - Continues with development tools to complete the construction
 - **Interface validation, focuses on**
 - The ability of the interface to implement every user task correctly, to accommodate all task variations, and to achieve all general user requirements

The Golden Rules of User Interface Design (by Theo Mandel'1997)

Rule 1. Place the User in Control

- Define interaction modes in a way that does not force a user into unnecessary or undesired actions
 - The user shall be able to enter and exit a mode with little or no effort (e.g., spell check ☺ edit text ☺ spell check)
- Provide for flexible interaction
 - The user shall be able to perform the same action via keyboard commands, mouse movement, or voice recognition
- Allow user interaction to be interruptible and "undo"able
 - The user shall be able to easily interrupt a sequence of actions to do something else (without losing the work that has been done so far)
 - The user shall be able to ^(More on next slide) "undo" any action

Place the User in Control (continued)

- Streamline interaction as skill levels advance and allow the interaction to be customized
 - The user shall be able to use a macro mechanism to perform a sequence of repeated interactions and to customize the interface
- Hide technical internals from the casual user
 - The user shall not be required to directly use operating system, file management, networking. etc., commands to perform any actions. Instead, these operations shall be hidden from the user and performed "behind the scenes" in the form of a real-world abstraction
- Design for direct interaction with objects that appear on the screen
 - The user shall be able to manipulate objects on the screen in a manner similar to what would occur if the object were a physical thing (e.g., stretch a rectangle, press a button, move a slider)

Rule 2. Reduce the User's Memory Load

- Reduce demand on short-term memory
 - The interface shall reduce the user's requirement to remember past actions and results by providing visual cues of such actions
- Establish meaningful defaults
 - The system shall provide the user with default values that make sense to the average user but allow the user to change these defaults
 - The user shall be able to easily reset any value to its original default value
- Define shortcuts that are intuitive
 - The user shall be provided **mnemonics** (i.e., control or alt combinations) that tie easily to the action in a way that is easy to remember such as the first letter

Reduce the User's Memory Load (continued)

- The visual layout of the interface should be based on a real world metaphor
 - The screen layout of the user interface shall contain well-understood visual cues that the user can relate to real-world actions
- Disclose information in a progressive fashion
 - When interacting with a task, an object or some behavior, the interface shall be organized hierarchically by moving the user progressively in a step-wise fashion from an abstract concept to a concrete action (e.g., text format options → format dialog box)

The more a user has to remember, the more error-prone interaction with the system will be

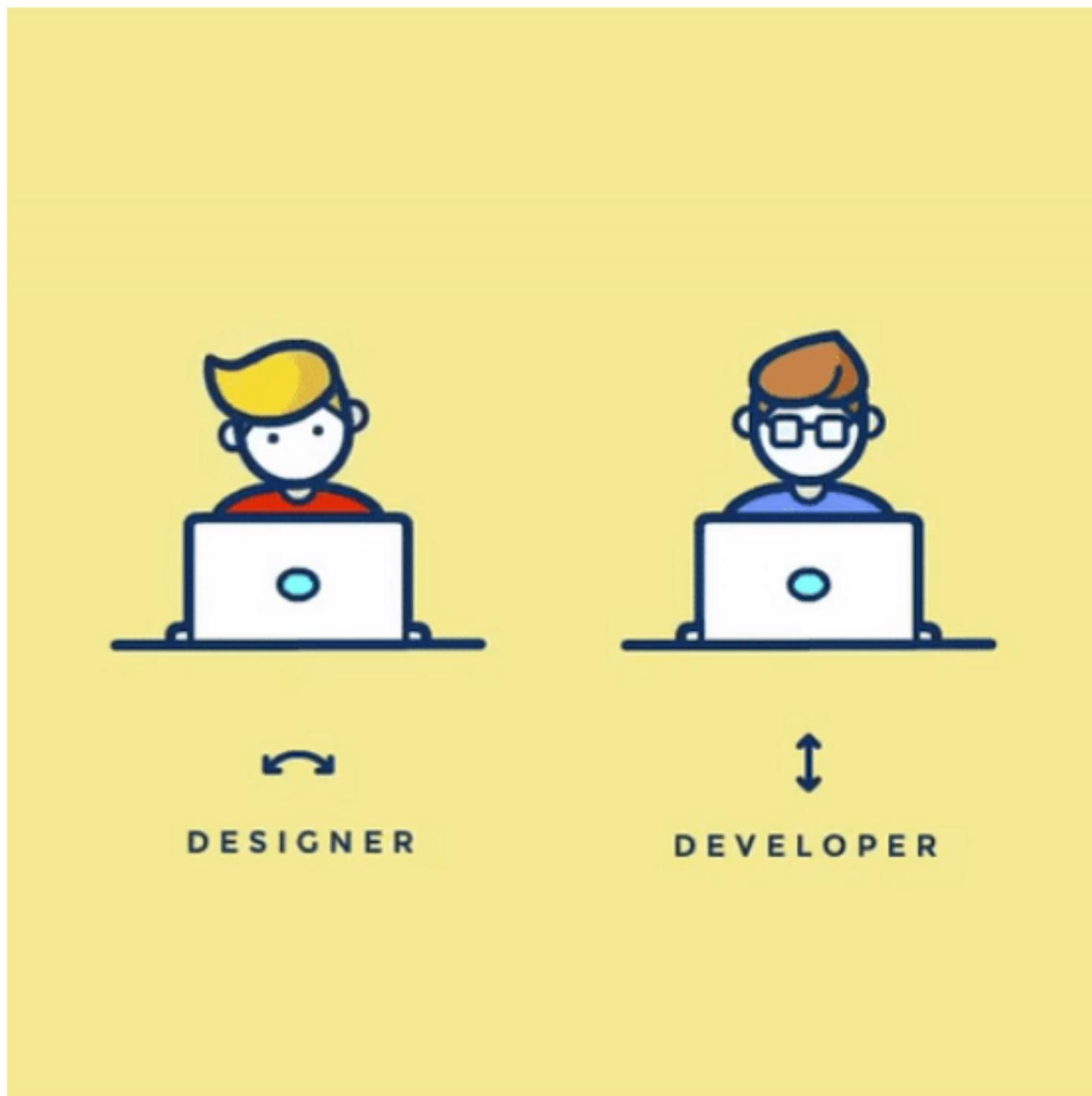
Rule 3. Make the Interface Consistent

- The interface should present and acquire information in a consistent fashion
 - All visual information shall be organized according to a design standard that is maintained throughout all screen displays
 - Input mechanisms shall be constrained to a limited set that is used consistently throughout the application
 - Mechanisms for navigating from task to task shall be consistently defined and implemented
- Allow the user to put the current task into a meaningful context
 - The interface shall provide indicators (e.g., window titles, consistent color coding) that enable the user to know the context of the work at hand
 - The user shall be able to determine where he has come from and what alternatives exist for a transition to a new task

Make the Interface Consistent (continued)

- **Maintain consistency across a family of applications**
 - A set of applications performing complimentary functionality shall all implement the same design rules so that consistency is maintained for all interaction
- **If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so**
 - Once a particular interactive sequence has become a de facto standard (e.g., alt-S to save a file), the application shall continue this expectation in every part of its functionality.

Designer or Developer?



UI Analysis and Design - Reconciling Four Different Models

Introduction

- Four different models come into play when a user interface is analyzed and designed
 - **User profile model** – Established by a human engineer or software engineer
 - **Design model** – Created by a software engineer
 - **Implementation model** – Created by the software programmers
 - **User's mental model** – Developed by the user when interacting with the application
- The role of the interface designer is to reconcile these differences and derive a consistent representation of the interface

User Profile Model

Jeff Patton –

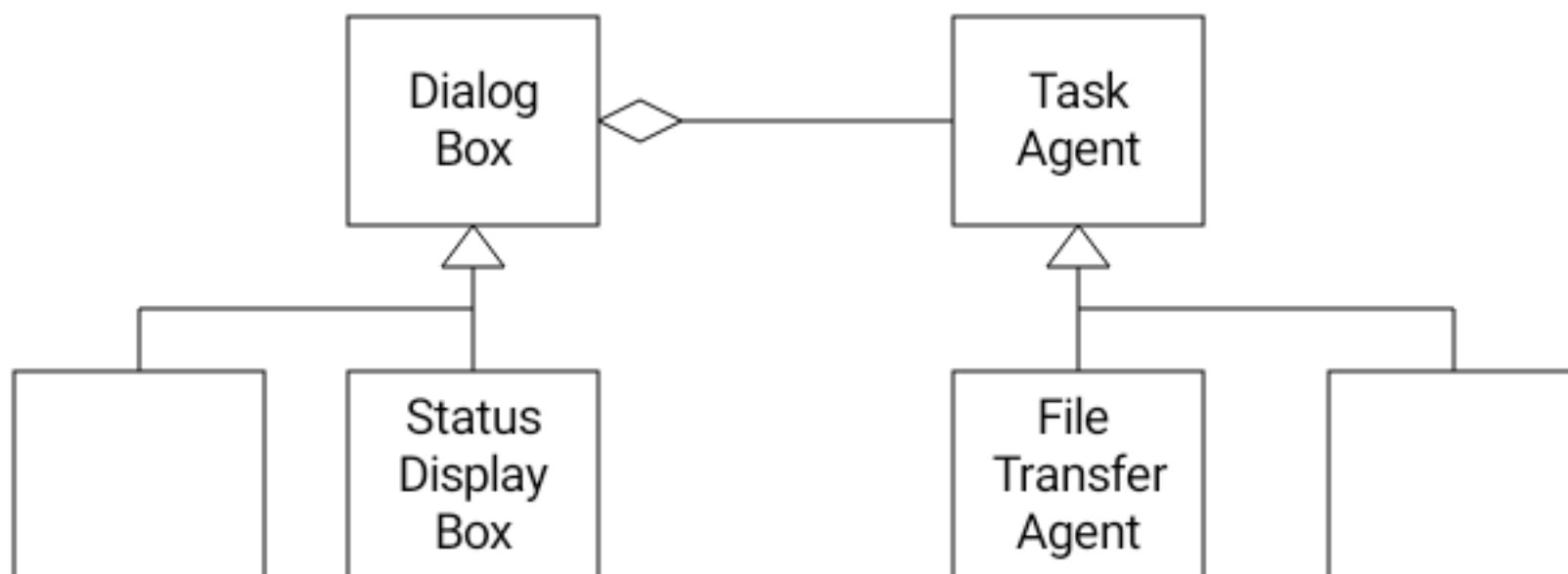
“The truth is, designers and developers—myself included—often think about users. However, in the absence of a strong mental model of specific users, we self-substitute. Self substitution isn’t user centric—it’s self-centric”.

User Profile Model

- Establishes the profile of the end-users of the system
 - Based on age, gender, physical abilities, education, cultural or ethnic background, motivation, goals, and personality
- Considers syntactic knowledge of the user
 - The mechanics of interaction that are required to use the interface effectively
- Considers semantic knowledge of the user
 - The underlying sense of the application; an understanding of the functions that are performed, the meaning of input and output, and the objectives of the system
- Categorizes users as
 - Novices
 - No syntactic knowledge of the system, little semantic knowledge of the application, only general computer usage
 - Knowledgeable, intermittent users
 - Reasonable semantic knowledge of the system, low recall of syntactic information to use the interface
 - Knowledgeable, frequent users
 - Good semantic and syntactic knowledge (i.e., power user), look for shortcuts and abbreviated modes of operation

Design Model

- Derived from the analysis model of the requirements
- Incorporates data, architectural, interface, and procedural representations of the software
- Constrained by information in the requirements specification that helps define the user of the system
- Normally is incidental to other parts of the design model
 - But in many cases it is as important as the other parts



Implementation Model

Know the user, know the tasks

“... pay attention to what users do, not what they say.” – Jakob Nielsen

- Consists of **the look and feel of the interface** combined with all supporting information (books, videos, help files) that describe system syntax and semantics
- Strives to agree with the user's mental model; users then feel comfortable with the software and use it effectively
- Serves as a translation of the design model by providing a realization of the information contained in the user profile model and the user's mental model

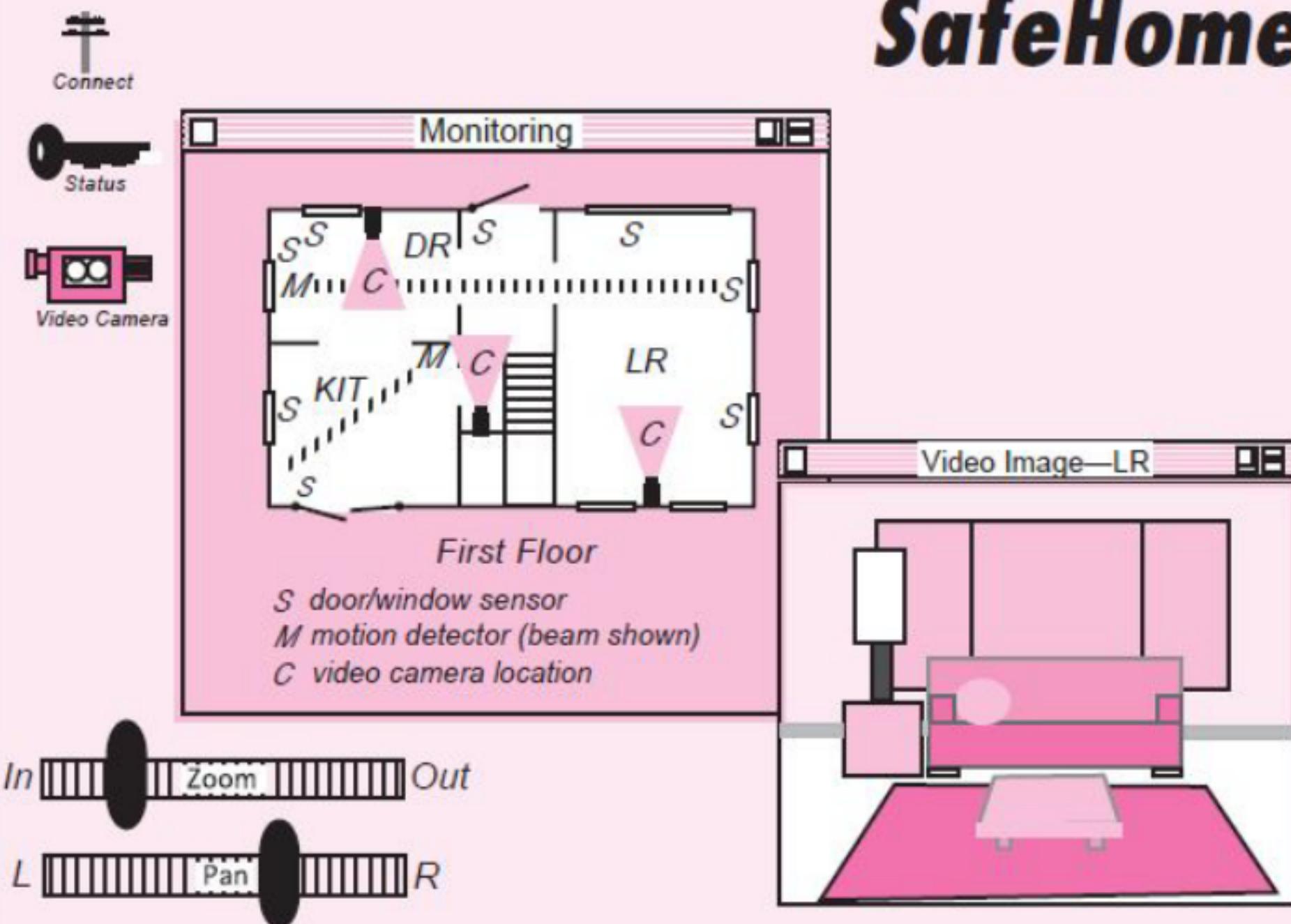
User's Mental Model

- Often called the [user's system perception](#)
- Consists of the image of the system that users carry in their heads
- Accuracy of the description depends upon the user's profile and overall familiarity with the software in the application domain

Example: A user who understands word processors fully but has worked with the specific word processor only once might actually be able to provide a more complete description of its function than the novice who has spent weeks trying to learn the system.

Access Configure System Status View Monitoring

SafeHome



User Interface Analysis

Elements of the User Interface

- To perform user interface analysis, the practitioner needs to study and understand four elements
 - The users who will interact with the system through the interface
 - The tasks that end users must perform to do their work
 - The content that is presented as part of the interface
 - The work environment in which these tasks will be conducted

User Analysis

- The analyst strives to get the end user's mental model and the design model to converge by understanding
 - The users themselves
 - How these people use the system
- Information can be obtained from
 - User interviews with the end users ([Do it for your current project](#))
 - Sales input from the sales people who interact with customers and users on a regular basis
 - Marketing input based on a market analysis to understand how different population segments might use the software
 - Support input from the support staff who are aware of what works and what doesn't, what users like and dislike, what features generate questions, and what features are easy to use
- A set of questions should be answered during user analysis (see next slide)

User Analysis Questions

- Are the users trained professionals, technicians, clerical or manufacturing workers?
- What level of formal education does the average user have?
- Are the users capable of learning on their own from written materials or have they expressed a desire for classroom training?
- Are the users expert typists or are they keyboard phobic?
- What is the age range of the user community?
- Will the users be represented predominately by one gender?
- How are users compensated for the work they perform or are they volunteers?
- Do users work normal office hours, or do they work whenever the job is required?
- Is the software to be an integral part of the work users do, or will it be used only occasionally?
- What is the primary spoken language among users?
- What are the consequences if a user makes a mistake using the system?
- Are users experts in the subject matter that is addressed by the system?

User interaction scenario

Sona is a student of Religious Studies and is working on an essay on Indian architecture and how it has been influenced by religious practices. To help her understand this, she would like to access some pictures of details on notable buildings but can't find anything in her local library. She approaches the subject librarian to discuss her needs and he suggests some search terms that might be used. He also suggests some libraries in Chennai and Singapore that might have this material so they log on to the library catalogues and do some searching using these terms. They find some source material and place a request for photocopies of the pictures with architectural detail to be posted directly to Sona.

Task Analysis and Modeling

- Task analysis strives to know and understand
 - The work the user performs in specific circumstances
 - The tasks and subtasks that will be performed as the user does the work
 - The specific problem domain objects that the user manipulates as work is performed
 - The sequence of work tasks (i.e., the workflow)
 - The hierarchy of tasks
- Use cases
 - Show how an end user performs some specific work-related task
 - Enable the software engineer to extract tasks, objects, and overall workflow of the interaction
 - Helps the software engineer to identify additional helpful features

Content Analysis

- The display content may range from character-based reports, to graphical displays, to multimedia information
- Display content may be
 - Generated by components in other parts of the application
 - Acquired from data stored in a database that is accessible from the application
 - Transmitted from systems external to the application in question
- The format and aesthetics of the content (as it is displayed by the interface) needs to be considered
- A set of questions should be answered during content analysis (see next slide)

Content Analysis Guidelines

- Are various types of data assigned to consistent locations on the screen (e.g., photos always in upper right corner)?
- Are users able to customize the screen location for content?
- Is proper on-screen identification assigned to all content?
- Can large reports be partitioned for ease of understanding?
- Are mechanisms available for moving directly to summary information for large collections of data?
- Is graphical output scaled to fit within the bounds of the display device that is used?
- How is color used to enhance understanding?
- How are error messages and warnings presented in order to make them quick and easy to see and understand?

Work Environment Analysis [Hackos and Redish'98]

States: Software products need to be designed to fit into the work environment, otherwise they may be difficult or frustrating to use

- Factors to consider include
 - Type of lighting
 - Display size and height
 - Keyboard size, height and ease of use
 - Mouse type and ease of use
 - Surrounding noise
 - Space limitations for computer and/or user
 - Weather or other atmospheric conditions
 - Temperature or pressure restrictions
 - Time restrictions (when, how fast, and for how long)

User Interface Design steps

Introduction

- User interface design is an **iterative process**, where each iteration elaborate and refines the information developed in the preceding step
- General steps for user interface design
 - Using information developed during user interface analysis, define user interface objects and actions (operations)
 - Define events (user actions) that will cause the state of the user interface to change; model this behavior
 - Depict each interface state as it will actually look to the end user
 - Indicate how the user interprets the state of the system from information provided through the interface
- During all of these steps, the designer must
 - **Always follow the three golden rules** of user interfaces
 - Model how the interface will be implemented
 - Consider the computing environment (e.g., display technology, operating system, development tools) that will be used

Interface Objects and Actions

- Interface objects and actions are obtained from a grammatical parse of the use cases and the software problem statement
- Interface objects are categorized into types: source, target, and application
 - A source object is dragged and dropped into a target object such as to create a hardcopy of a report
 - An application object represents application-specific data that are not directly manipulated as part of screen interaction such as a list
- After identifying objects and their actions, an interface designer performs screen layout which involves
 - Graphical design and placement of icons
 - Definition of descriptive screen text
 - Specification and titling for windows

Preliminary screen layout

Access Configure System Status View Monitoring

SafeHome

The main window displays a floor plan of a First Floor. The plan includes a central entrance area with a door labeled 'DR', a living room labeled 'LR', and a kitchen labeled 'KIT'. Various sensors and cameras are indicated: 'S' for door/window sensors, 'M' for motion detectors (represented by beams), and 'C' for video camera locations. A legend at the bottom left defines these symbols.

First Floor

S door/window sensor
M motion detector (beam shown)
C video camera location

In *Out*

L *R*

Monitoring

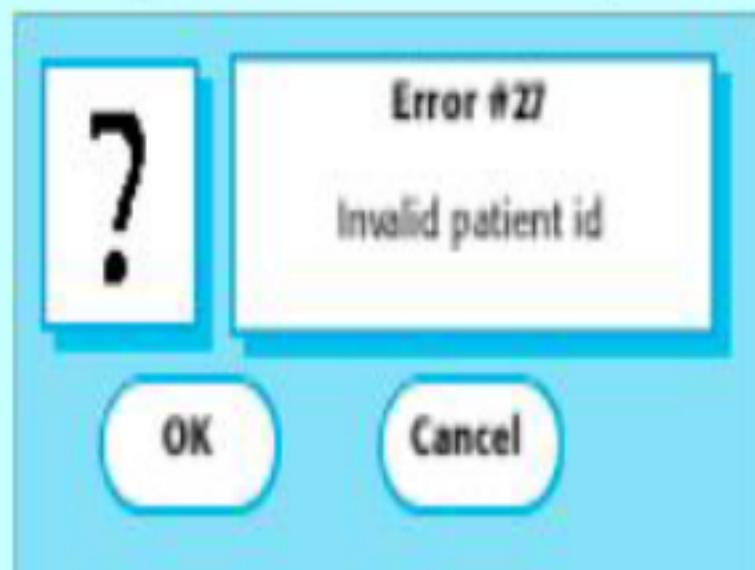
Video Image—LR

Design Issues to Consider

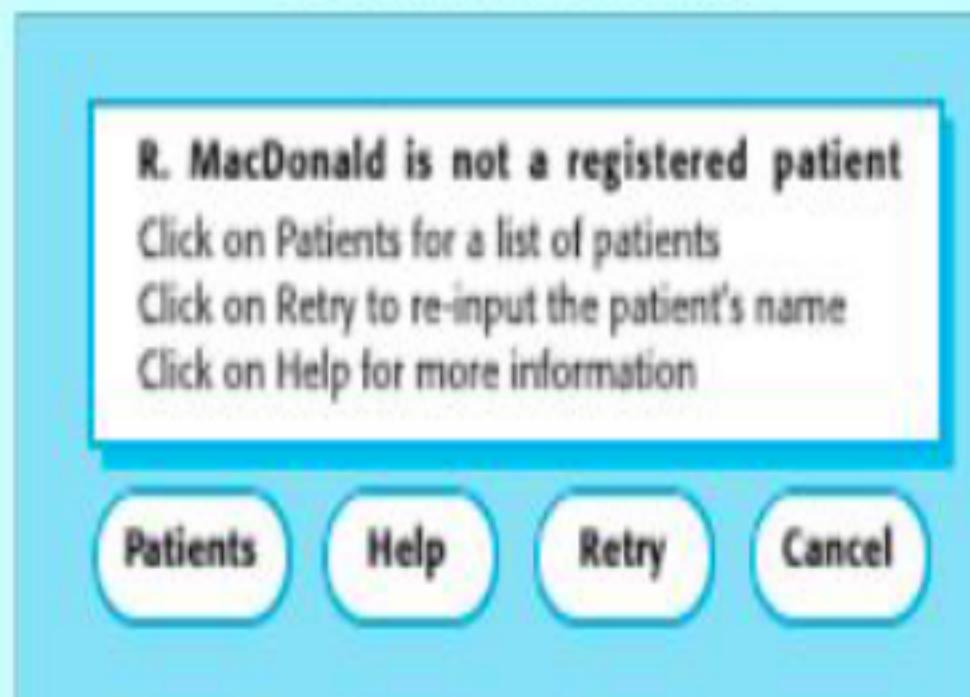
- Four common design issues usually surface in any user interface
 - System response time (both length and variability)
 - User help facilities
 - When is it available, how is it accessed, how is it represented to the user, how is it structured, what happens when help is exited
 - Error information handling (more on next slide)
 - How meaningful to the user, how descriptive of the problem
 - Menu and command labeling (more on upcoming slide)
 - Consistent, easy to learn, accessibility, internationalization
- Many software engineers do not address these issues until late in the design or construction process
 - This results in unnecessary iteration, project delays, and customer frustration

Avoiding Negative forms

System-oriented error message

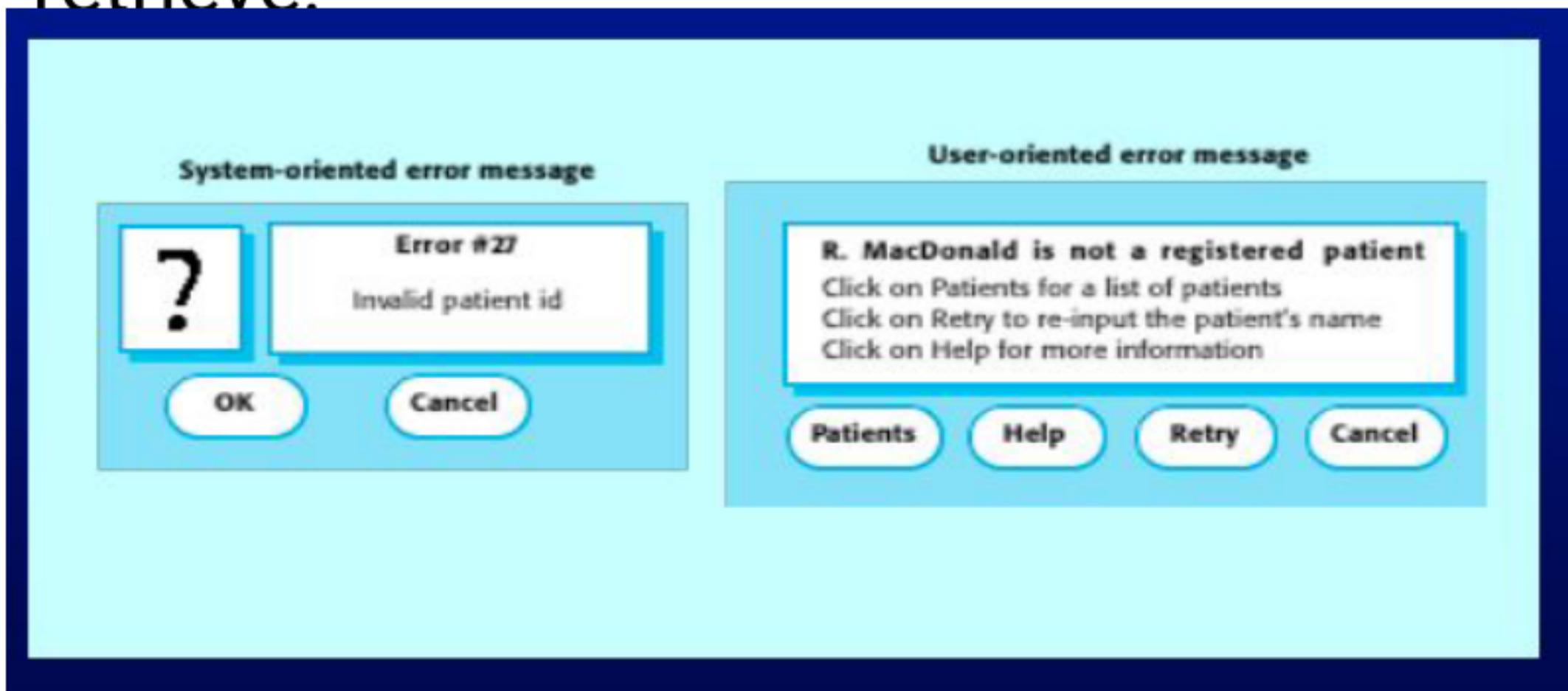


User-oriented error message



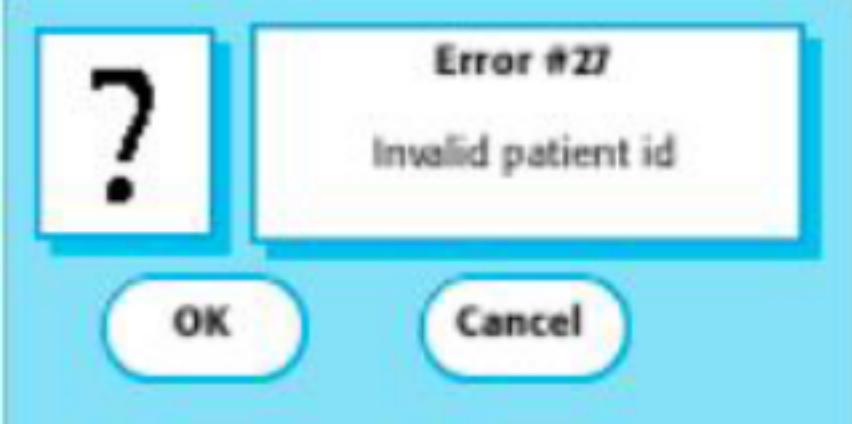
User error

- Assume that a nurse misspells the name of a patient whose records he is trying to retrieve.

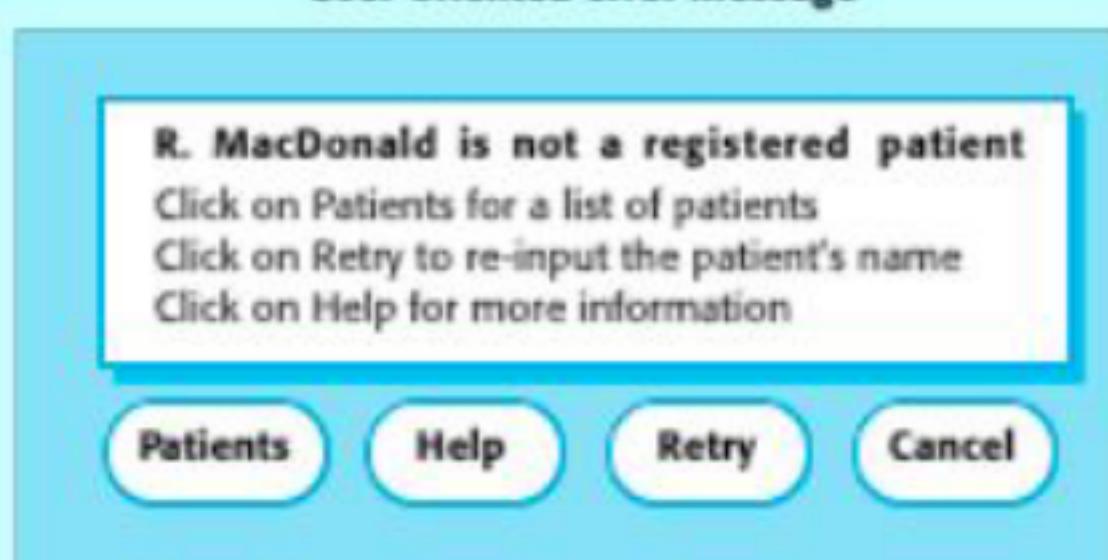


Good and bad message design

System-oriented error message



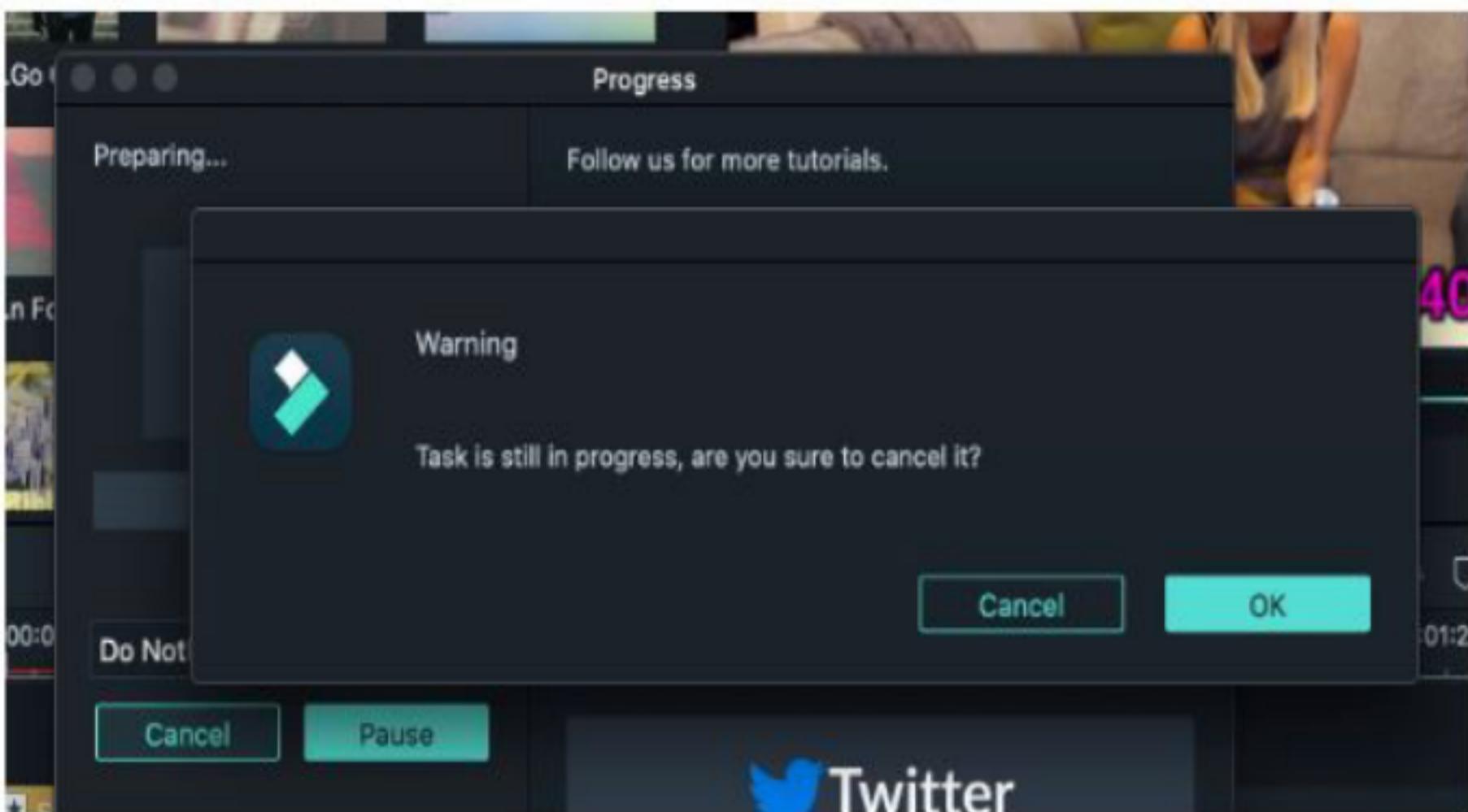
User-oriented error message

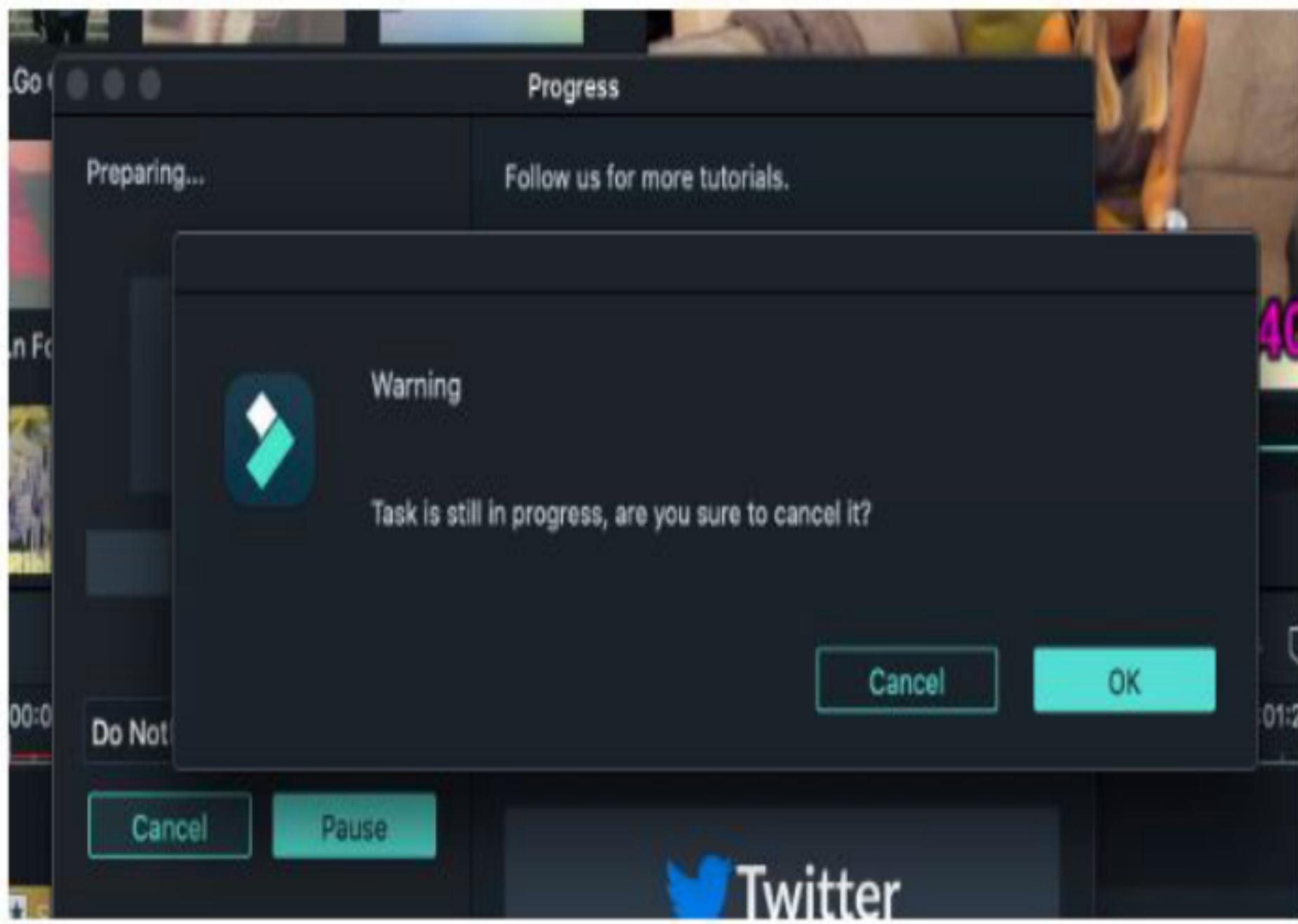


Guidelines for Error Messages

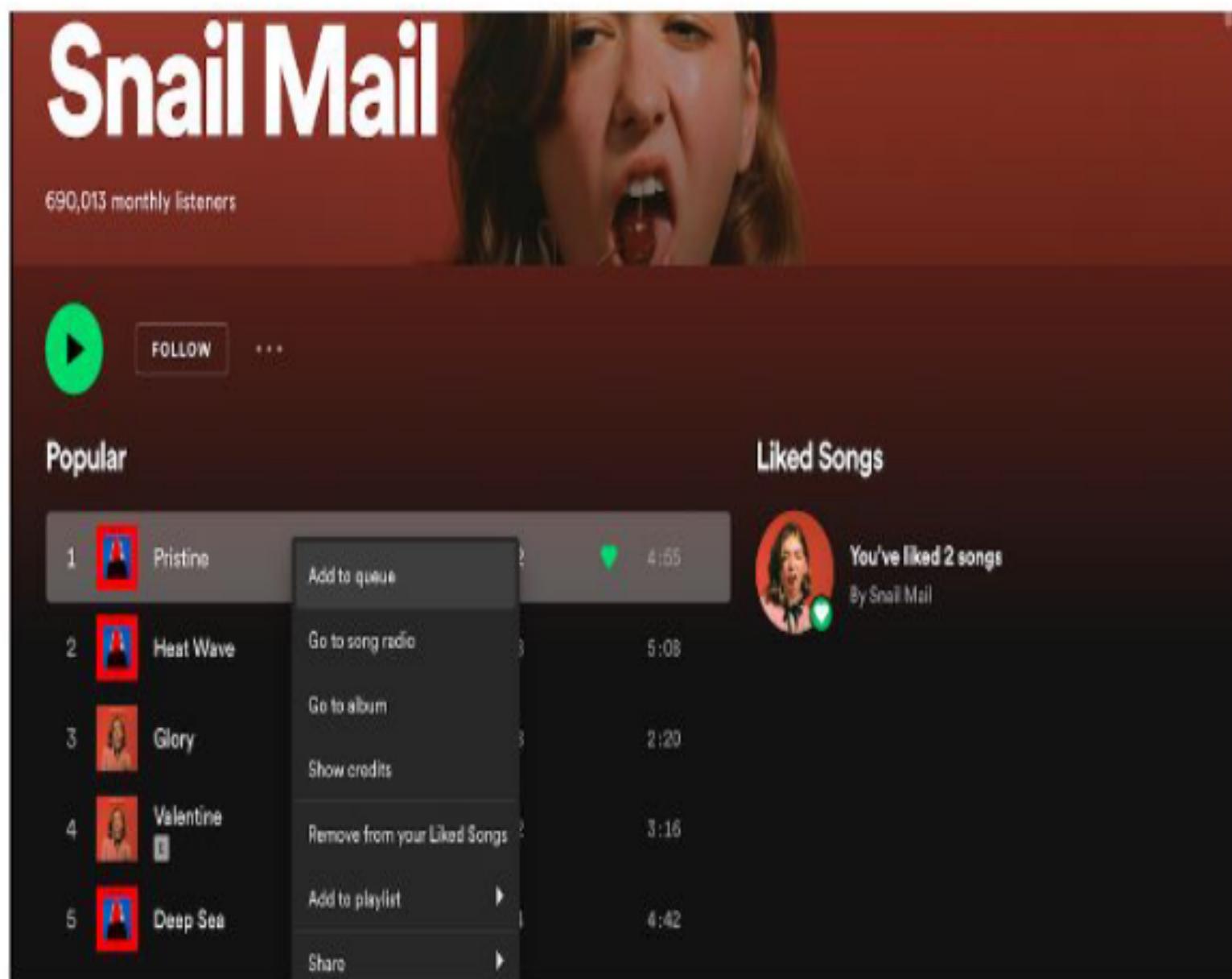
1. Keep language clear and concise
2. Keep user actions specific and logical
3. Avoid oops and whoops
4. Don't blame the user
5. Avoid ambiguity
6. Don't mock your users / Keep the jokes to a minimum
7. Avoid negative words
8. Write for humans
9. Don't write in ALL CAPS (and avoid exclamation marks)
10. Try to use inline validation

Terrible!





I feel like these are trick buttons for users



Instagram gives users two clear actions that go along with the message above them

Guidelines for Error Messages contd'

- The message should describe the problem in plain language that a typical user can understand
- The message should provide constructive advice for recovering from the error
- The message should indicate any negative consequences of the error (e.g., potentially corrupted data files) so that the user can check to ensure that they have not occurred (or correct them if they have)
- The message should be accompanied by an audible or visual cue such as a beep, momentary flashing, or a special error color
- The message should be non-judgmental
 - The message should never place blame on the user

An effective error message philosophy can do much to improve the quality of an interactive system and will significantly reduce user frustration when problems do occur

Questions for Menu Labeling and Typed Commands

- Will every menu option have a corresponding command?
- What form will a command take? A control sequence? A function key? A typed word?
- How difficult will it be to learn and remember the commands?
- What can be done if a command is forgotten?
- Can commands be customized or abbreviated by the user?
- Are menu labels self-explanatory within the context of the interface?
- Are submenus consistent with the function implied by a master menu item?

User Interface Evaluation

Design and Prototype Evaluation

- Before prototyping occurs, a number of evaluation criteria can be applied during design reviews to the design model itself
 - **The amount of learning required by the users**
 - Derived from the length and complexity of the written specification and its interfaces
 - **The interaction time and overall efficiency**
 - Derived from the number of user tasks specified and the average number of actions per task
 - **The memory load on users**
 - Derived from the number of actions, tasks, and system states
 - **The complexity of the interface and the degree to which it will be accepted by the user**
 - Derived from the interface style, help facilities, and error handling procedures

Design and Prototype Evaluation (continued)

- Prototype evaluation can range from an informal test drive to a formally designed study using statistical methods and questionnaires
- The prototype evaluation cycle consists of prototype creation followed by user evaluation and back to prototype modification until all user issues are resolved
- The prototype is evaluated for
 - Satisfaction of user requirements
 - Conformance to the three golden rules of user interface design
 - Reconciliation of the four models of a user interface

Webapp Interface Design

Dix says:

Where am I?

What can I do now?

Where have I been, where am I going?

Webapp Interface design principles and guidelines

First Impression

Bruce Tognazzi'2001 –

Anticipation

Communication

Consistency

Controlled autonomy

Efficiency

Requirement Elaboration - Sequence Diagrams and Collaboration Diagrams

Slides adapted from various web sources with grateful acknowledgement of the many others who made their course materials freely available online.

Interaction Diagrams

- UML Specifies a number of interaction diagrams to model dynamic aspects of the system
- Dynamic aspects of the system
 - Messages moving among objects/classes
 - Flow of control among objects
 - Sequences of events

Dynamic Diagram Types

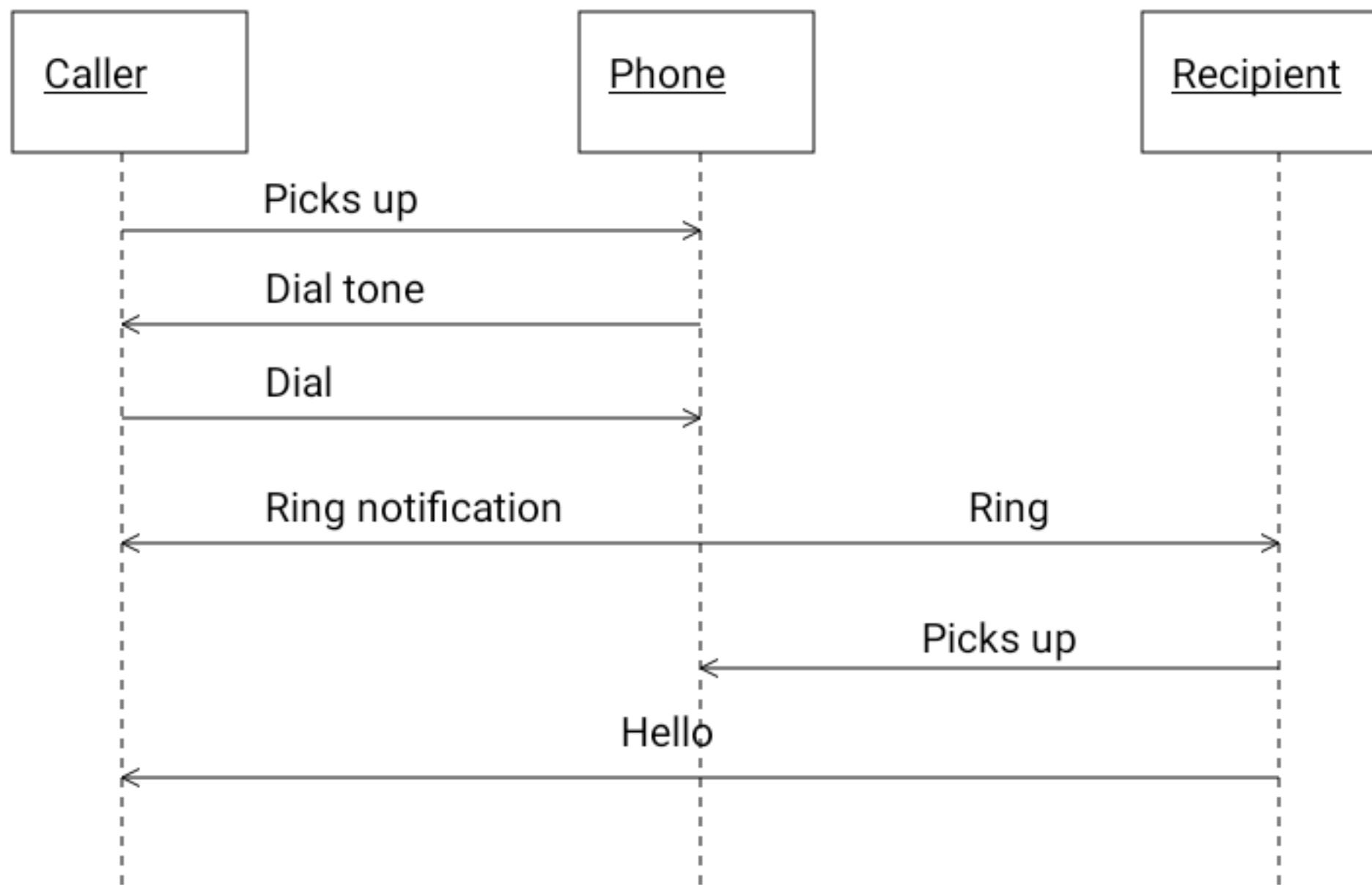
- **Interaction Diagrams** - Set of objects or roles and the messages that can be passed among them.
 - Sequence Diagrams - emphasize time ordering
 - Communication (Collaboration) Diagrams - emphasize structural ordering
- **State Diagrams**
 - State machine consisting of states, transitions, events and activities of an object
- **Activity & Swimlane Diagrams**
 - Emphasize and show flow of control among objects

Sequence Diagrams

- Describe the flow of messages, events, actions between objects
- Show concurrent processes and activations
- Show time sequences that are not easily depicted in other diagrams
- Typically used during analysis and design to document and understand the logical flow of your system

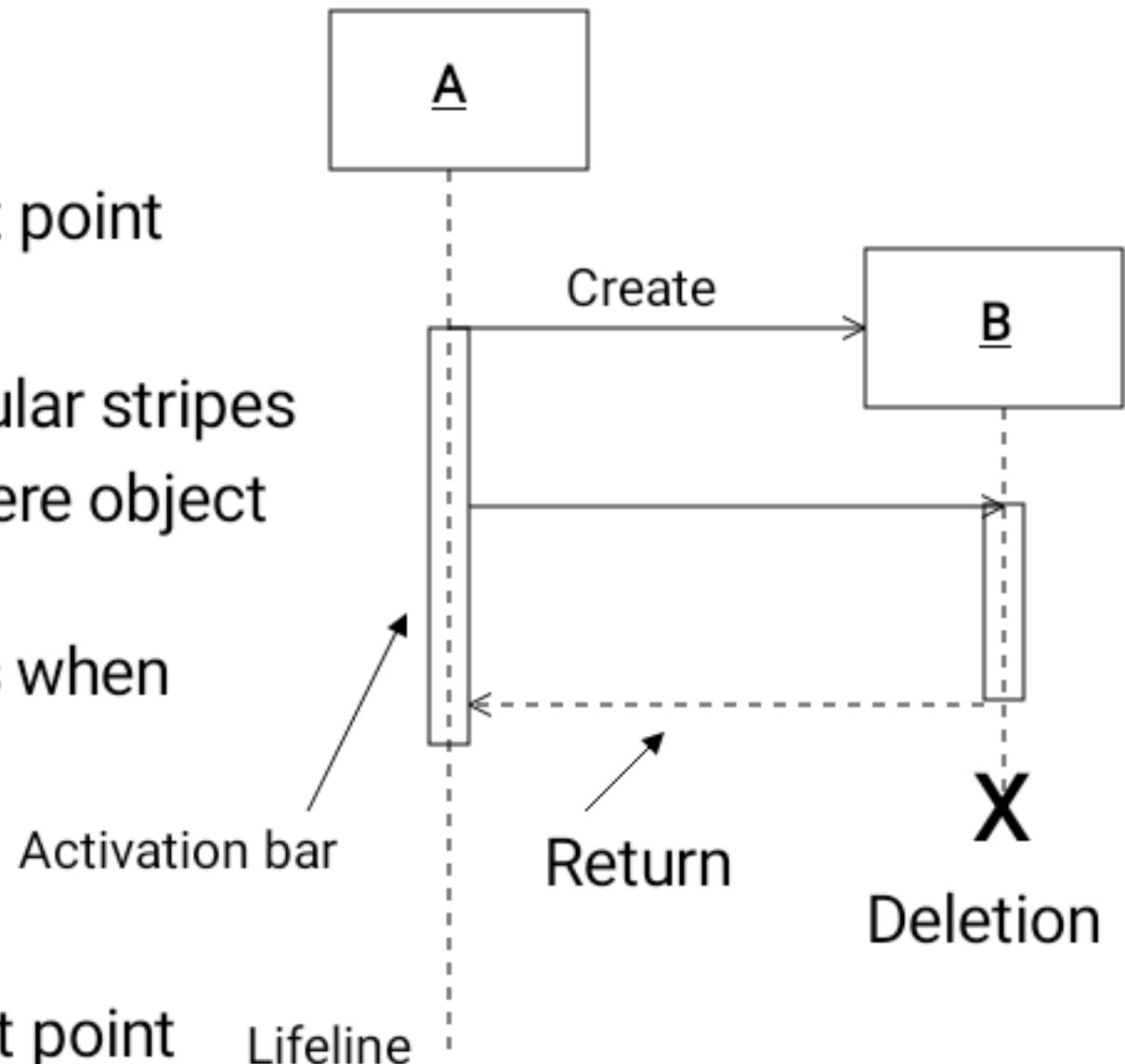
Emphasis on time ordering!

Sequence Diagram(make a phone call)

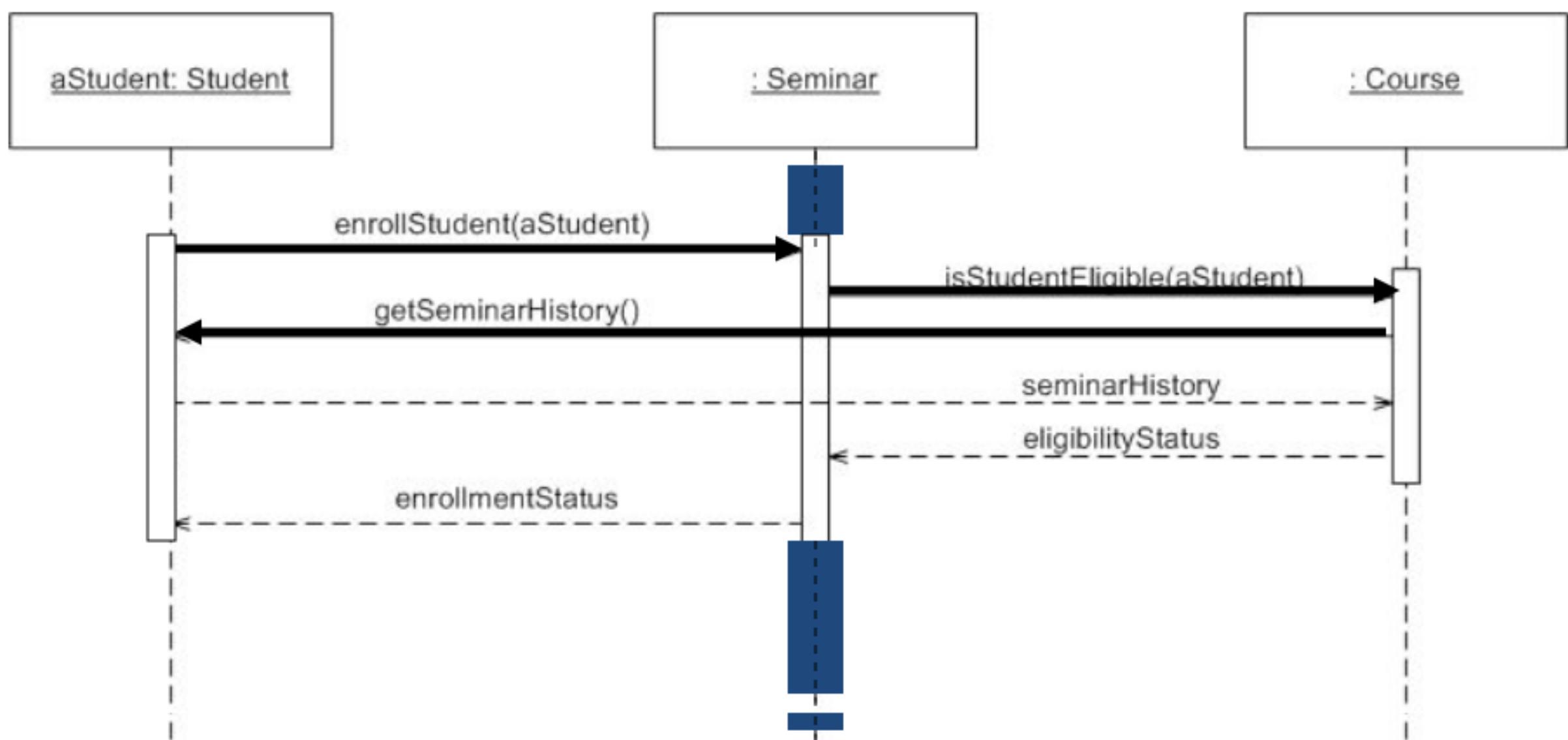


Sequence Diagrams – Object Life Spans

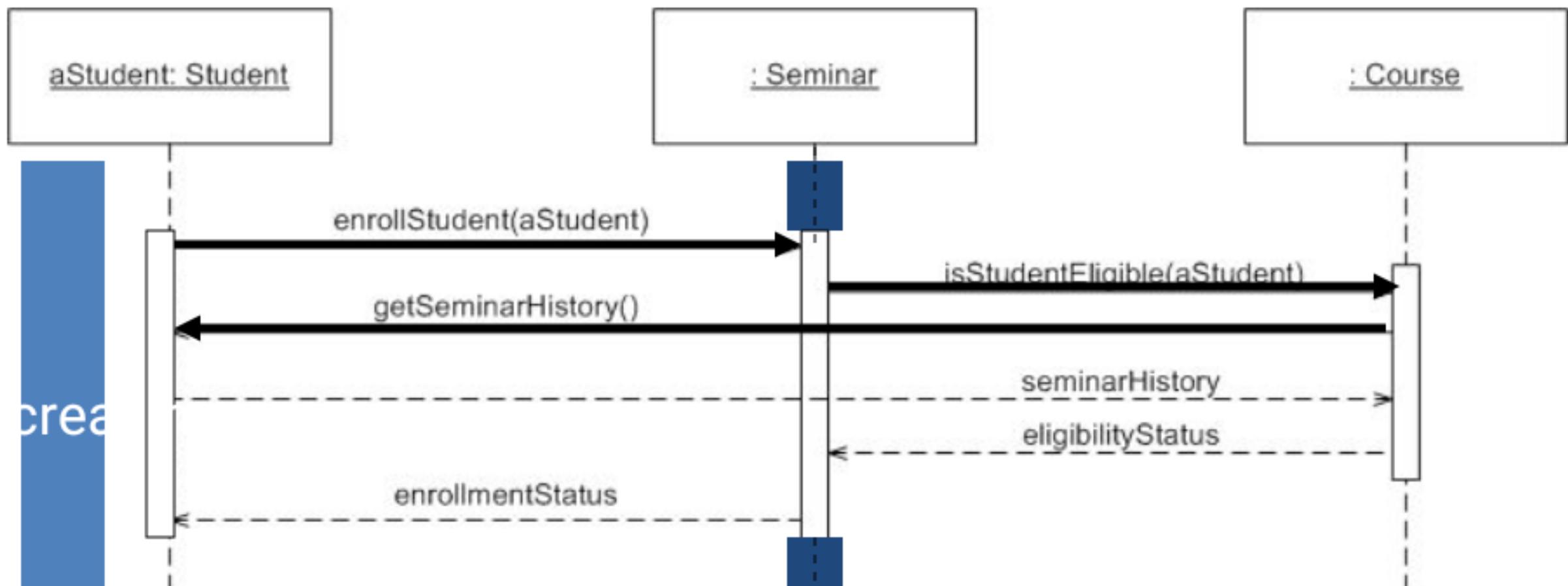
- Creation
 - ★ Create message
 - ★ Object life starts at that point
- Activation
 - ★ Symbolized by rectangular stripes
 - ★ Place on the lifeline where object is activated.
 - ★ Rectangle also denotes when object is deactivated.
- Deletion
 - ★ Placing an 'X' on lifeline
 - ★ Object's life ends at that point



Sequence Diagram



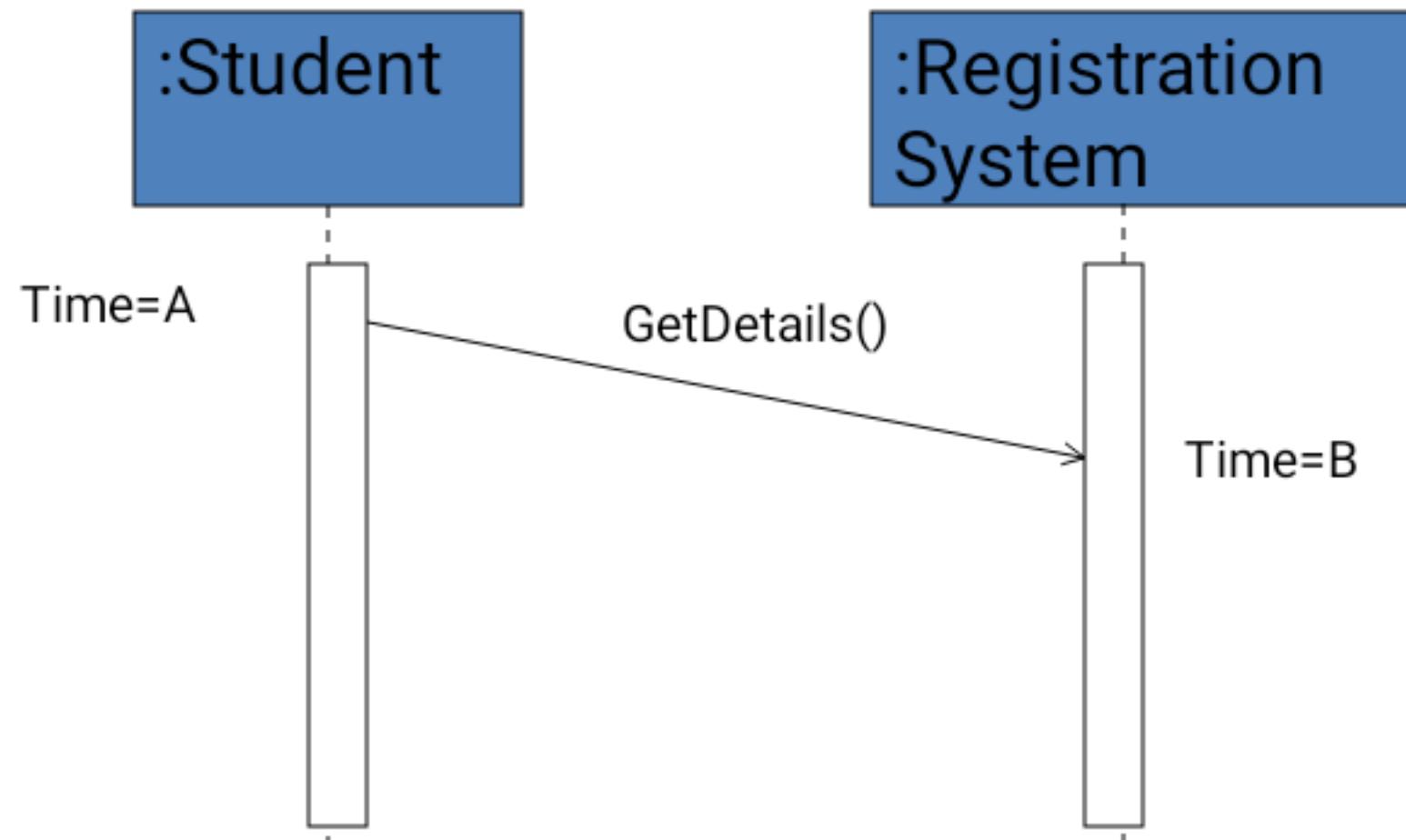
Sequence Diagram



All lines should be horizontal to indicate instantaneous actions. Additionally if Activity A happens before Activity B, Activity A must be above activity B

Lower = Later!

Diagonal Lines

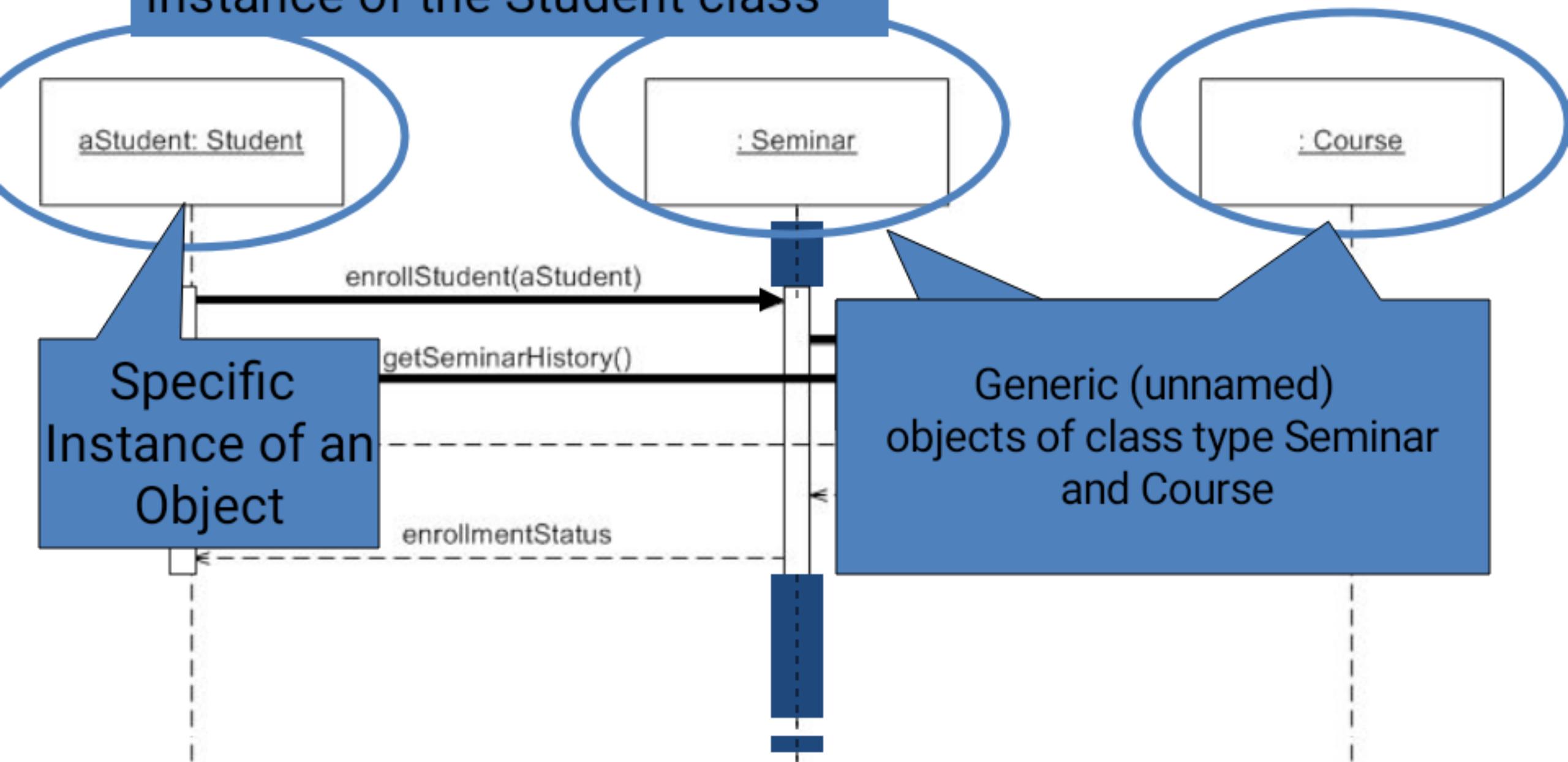


- What does this mean?

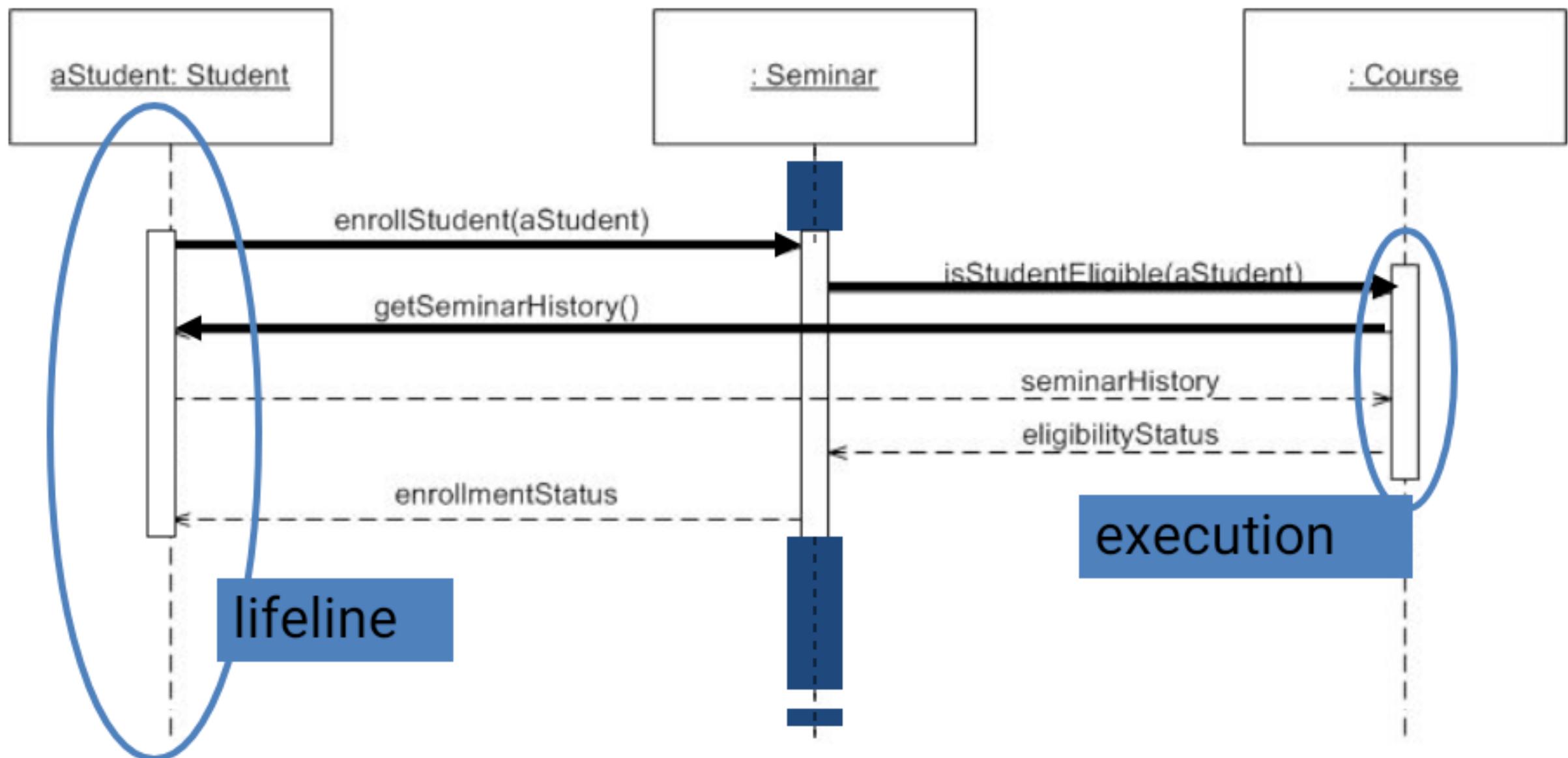
Do you typically care?

Components

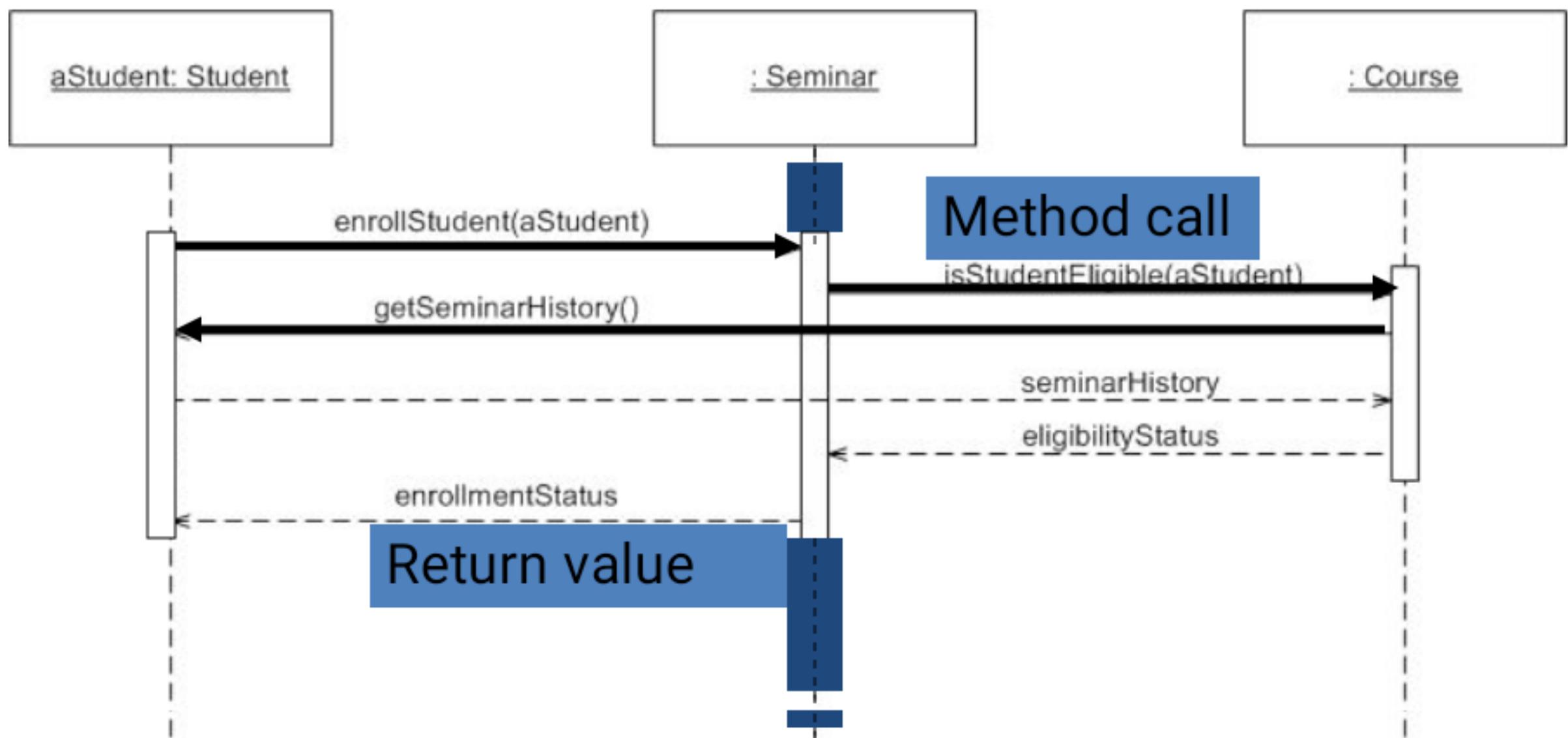
Objects: aStudent is a specific instance of the Student class



Components



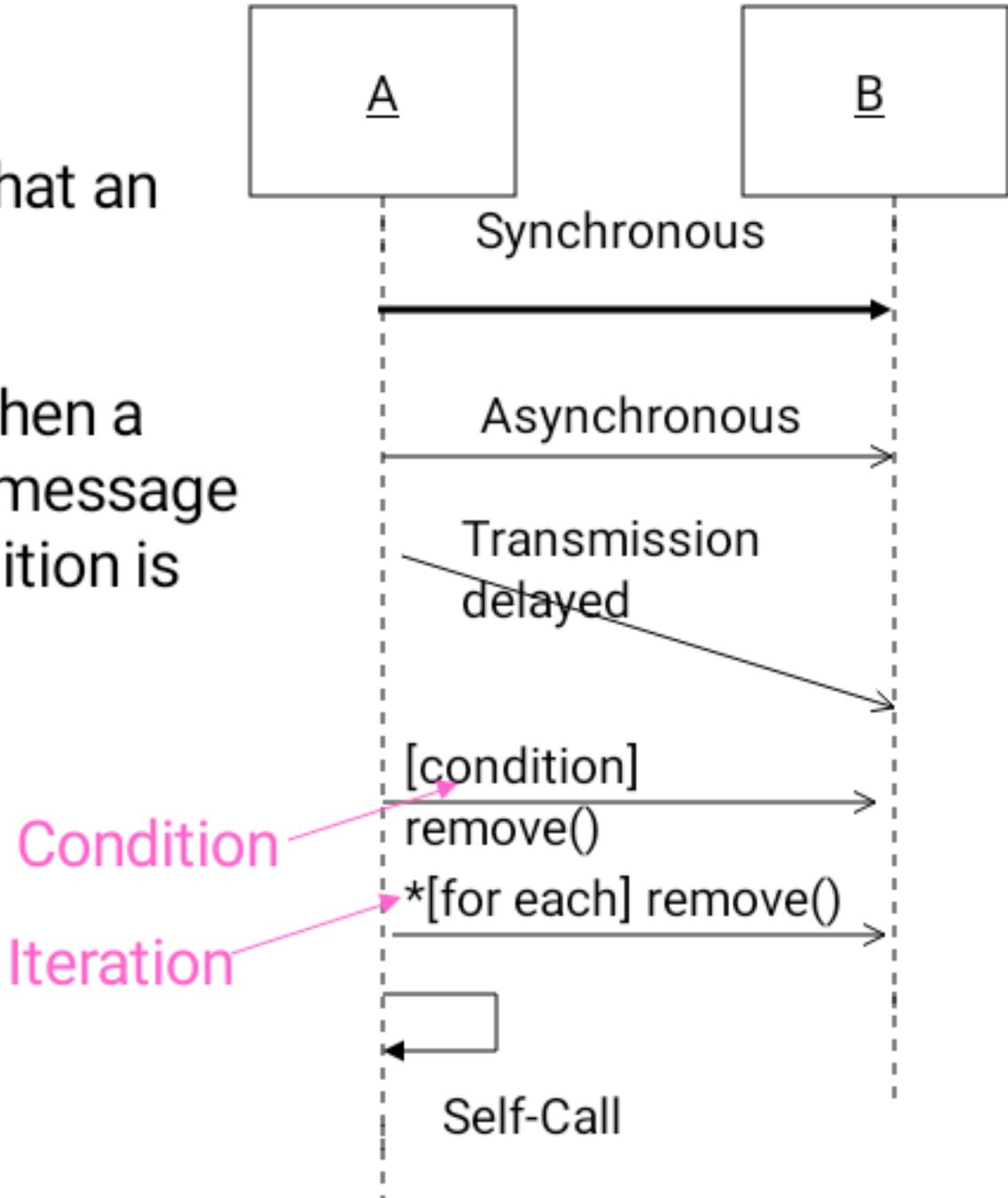
Components



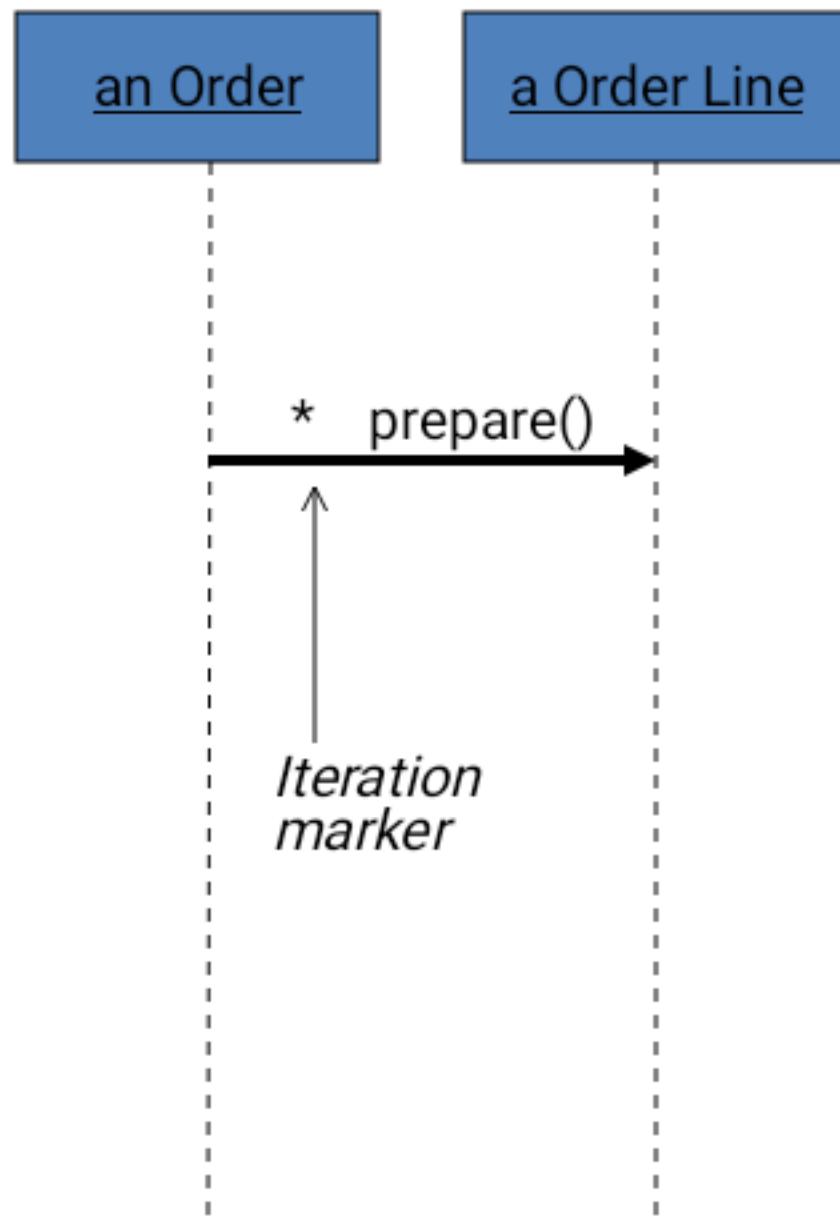
Sequence Diagram: Object interaction

Self-Call: A message that an Object sends to itself.

Condition: indicates when a message is sent. The message is sent only if the condition is true.

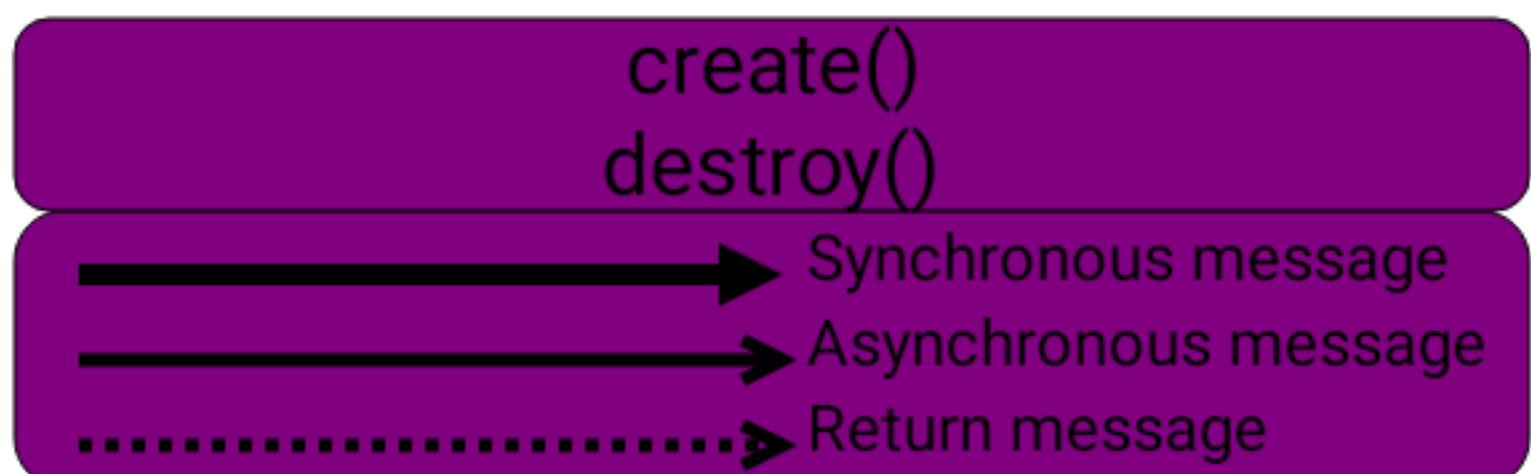
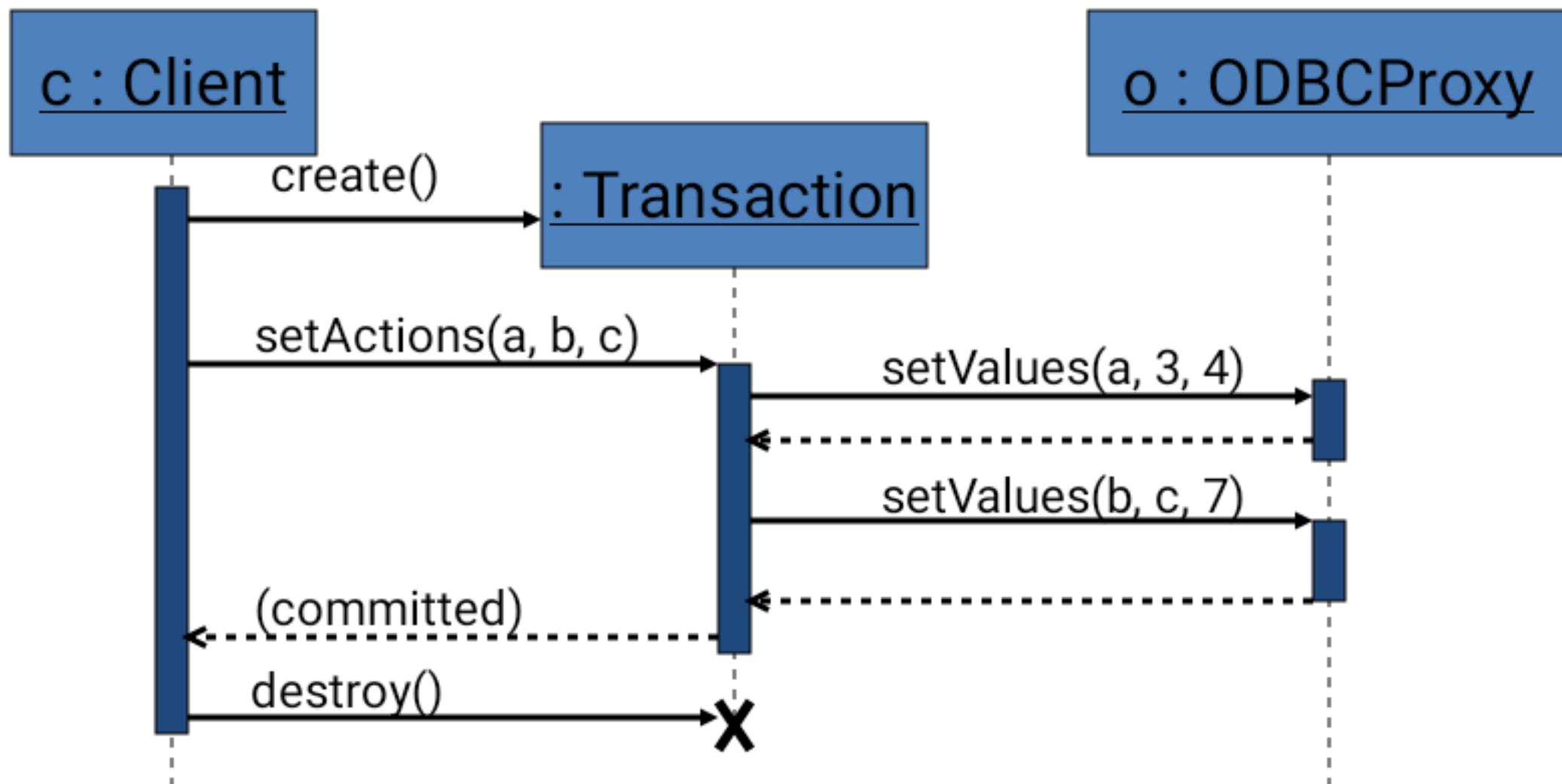


Sequence Diagram

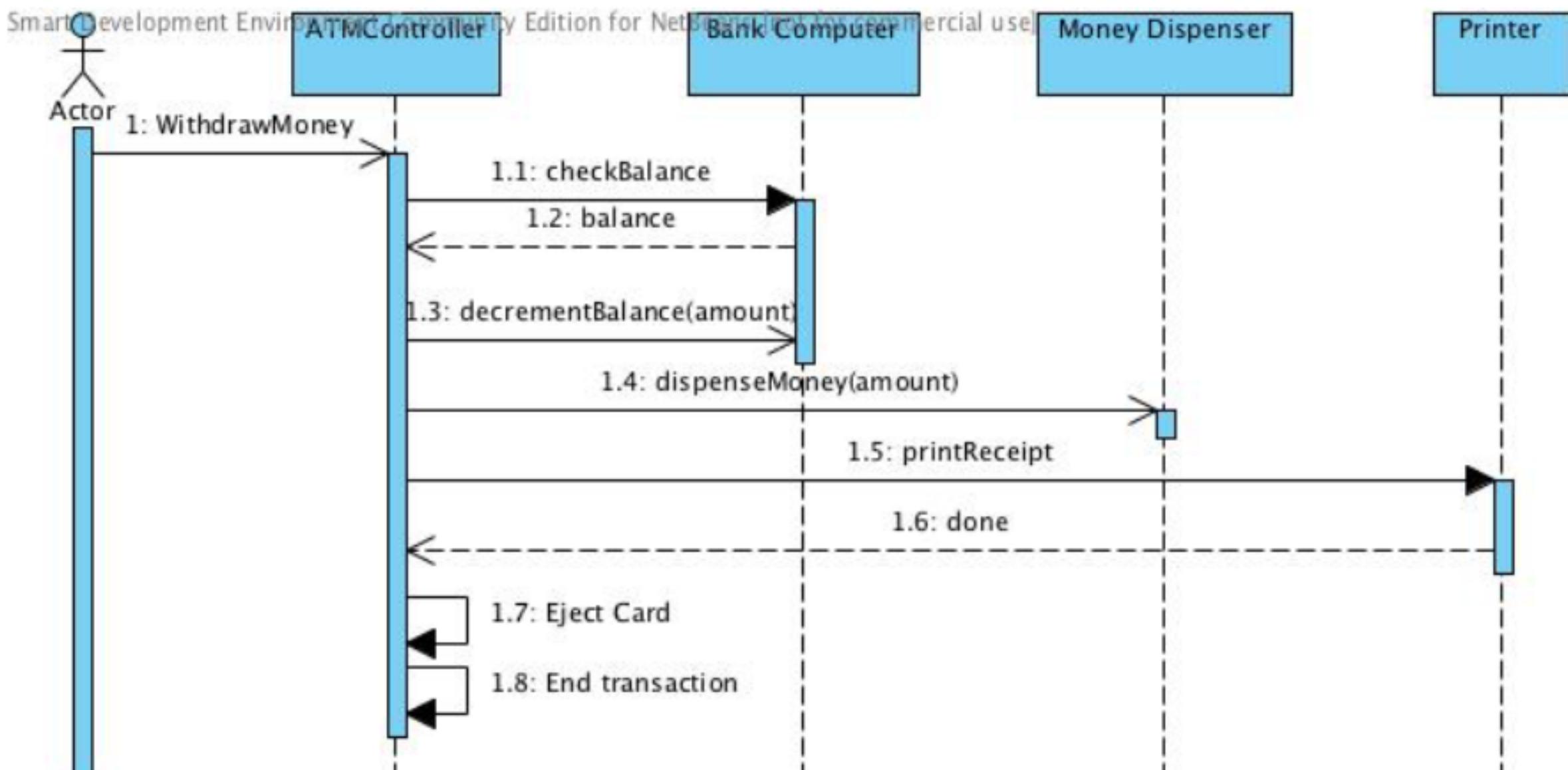


An iteration marker, such as * (as shown), or *[i = 1..n] , indicates that a message will be repeated as indicated.

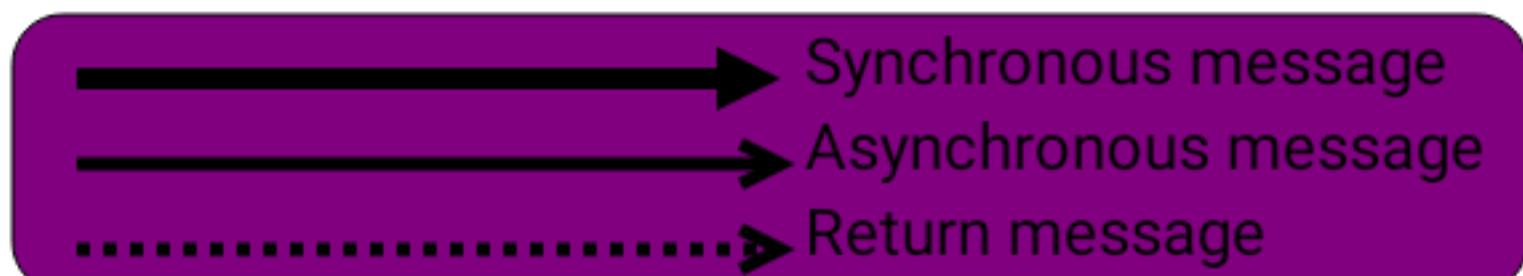
Components



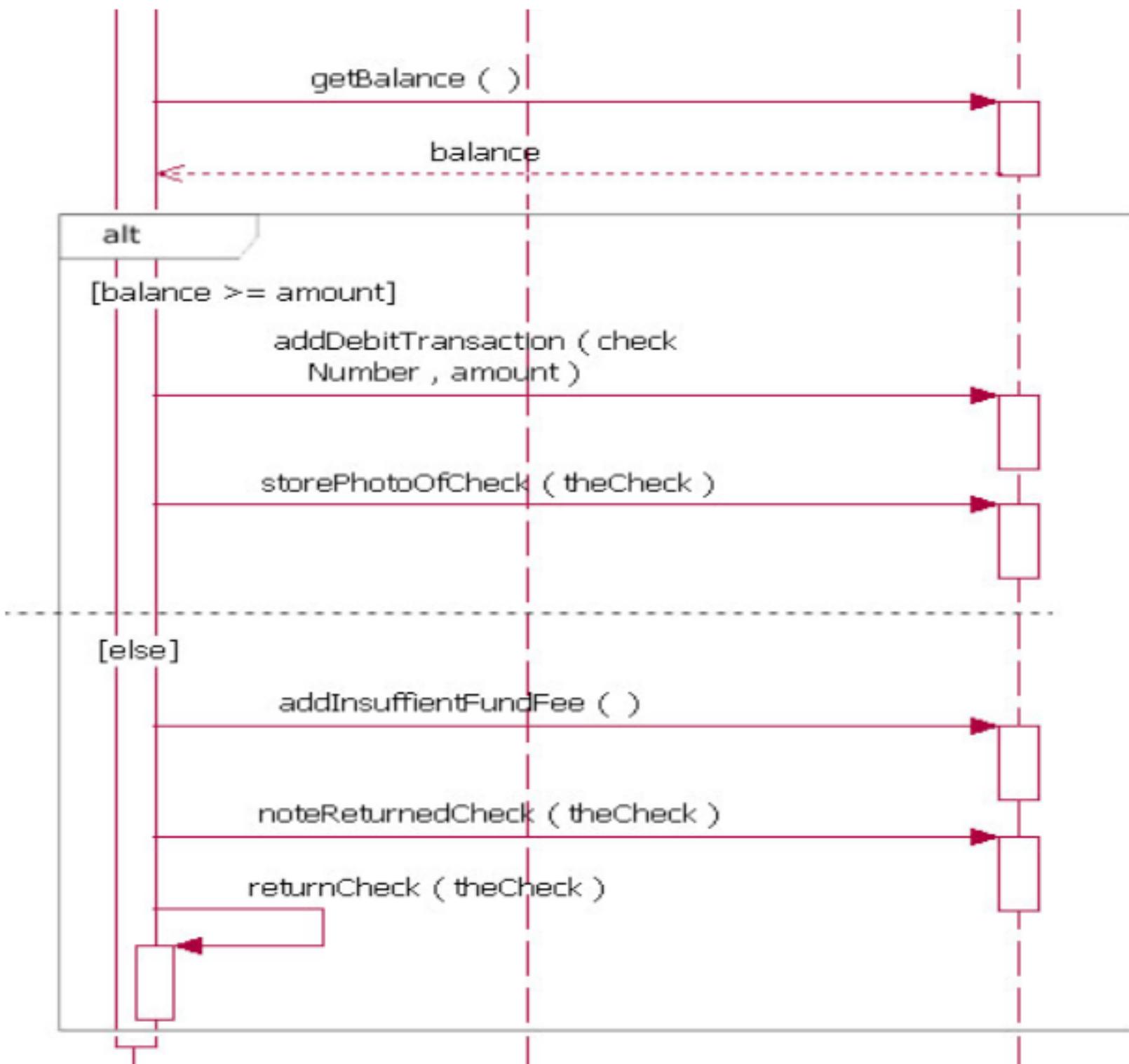
Async Message Example



There are problems here... what are they?



Components: alt/else



Rules of thumb

- Rarely use alt/else
 - These constructs complicate a diagram and make them hard to read/interpret.
 - Frequently it is better to create multiple simple diagrams
- Create sequence diagrams for use cases when it helps clarify and visualize a complex flow
- Remember: the goal of UML is communication and understanding

Summary

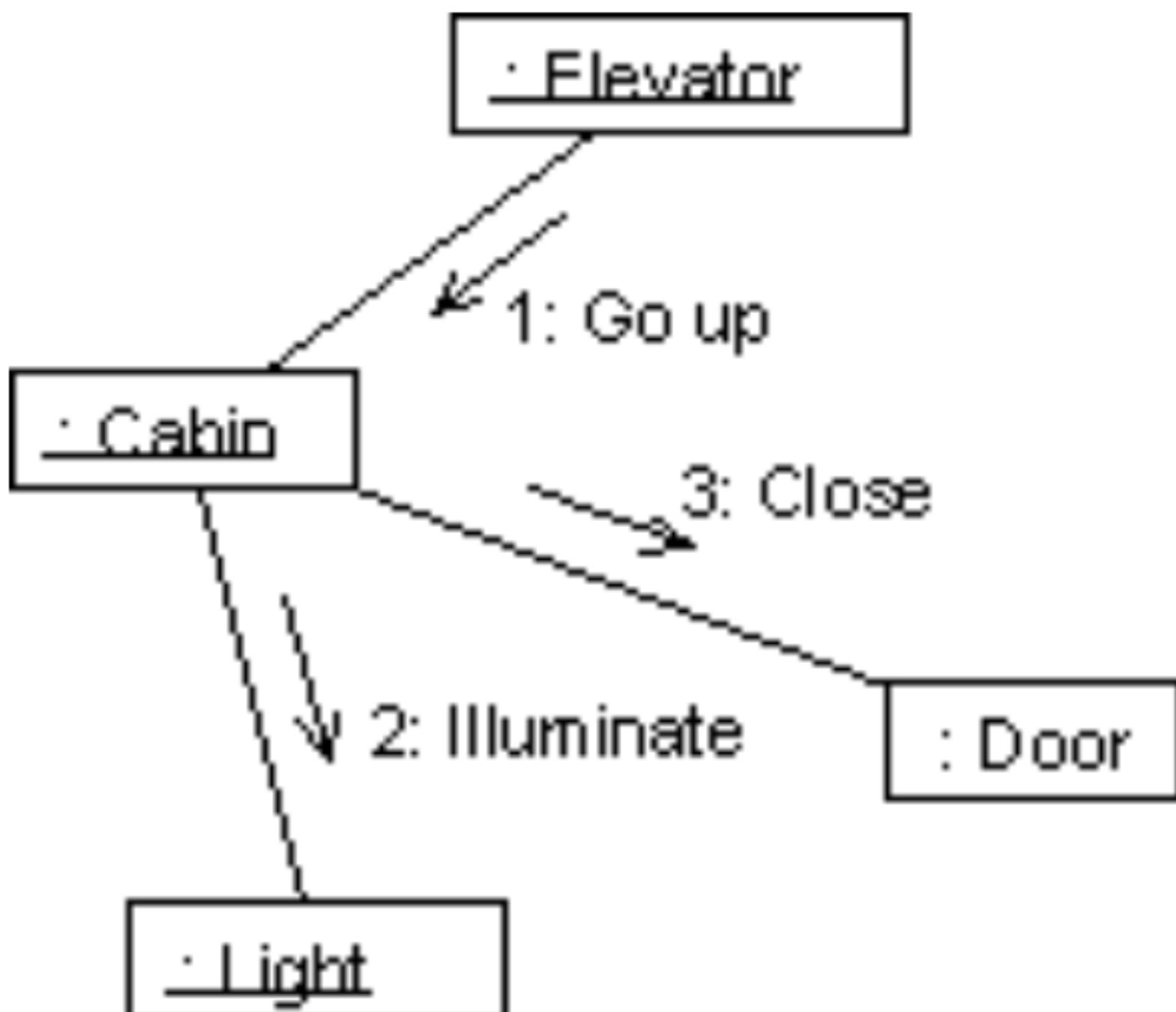
- Sequence diagrams model object interactions with an emphasis on time ordering
- Method call lines
 - Must be horizontal!
 - Vertical height matters! “Lower equals Later”
 - Label the lines
- Lifeline – dotted vertical line
- Execution bar – bar around lifeline when code is running
- Arrows
 - Synchronous call (you’re waiting for a return value) – triangle arrow-head
 - Asynchronous call (not waiting for a return) – open arrow-head
 - Return call – dashed line

Collaboration Diagram

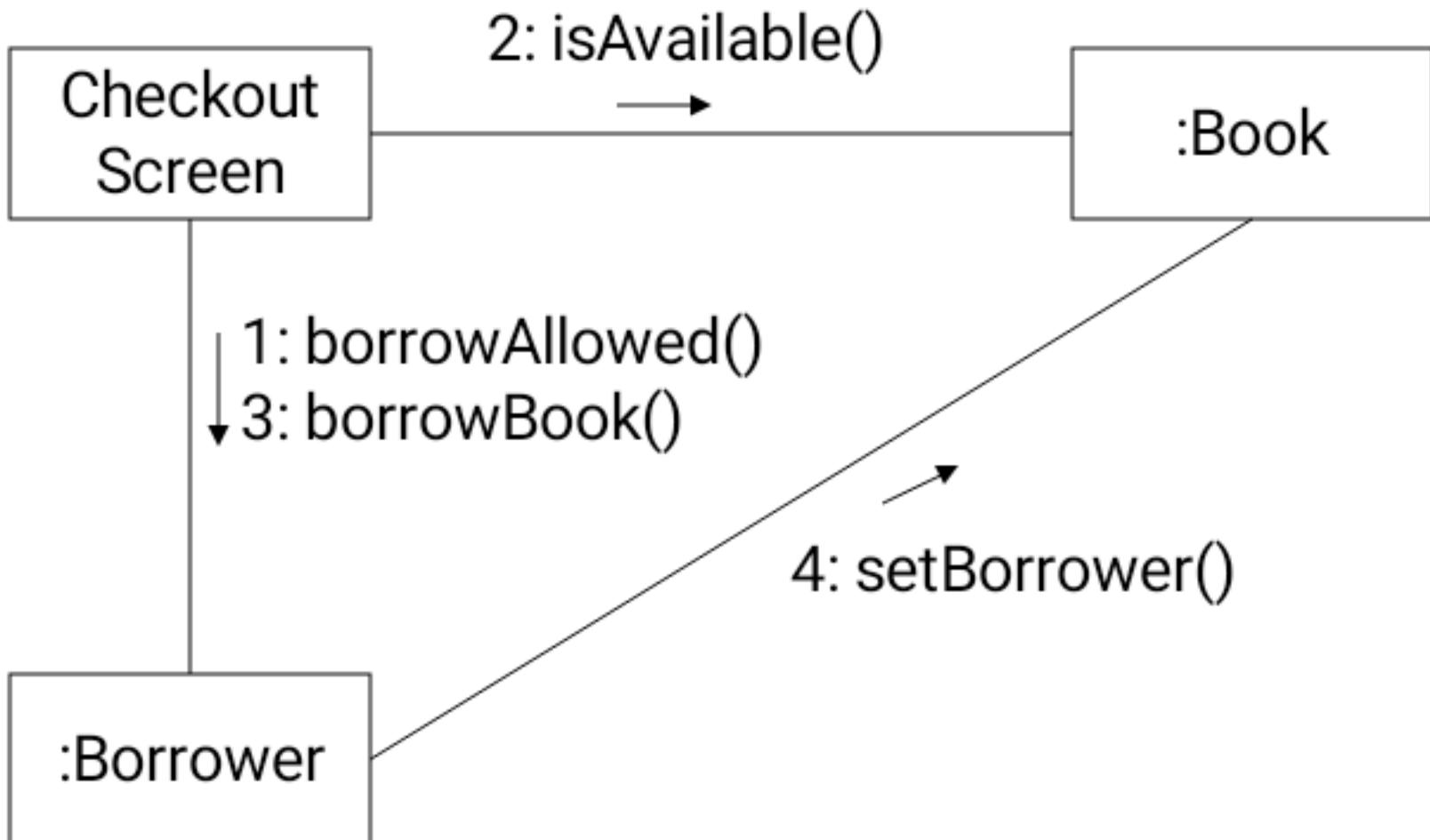
A collaboration diagram emphasizes the relationship of the objects that participate in an interaction. Unlike a sequence diagram, you **don't have to show the lifeline** of an object explicitly in a collaboration diagram. The sequence of events are indicated by sequence numbers preceding messages.

Object identifiers are of the form *objectName* : *className*, and either the *objectName* or the *className* can be omitted, and the placement of the colon indicates either an *objectName* : , or a :*className*.

Ex. Collaboration Diagram



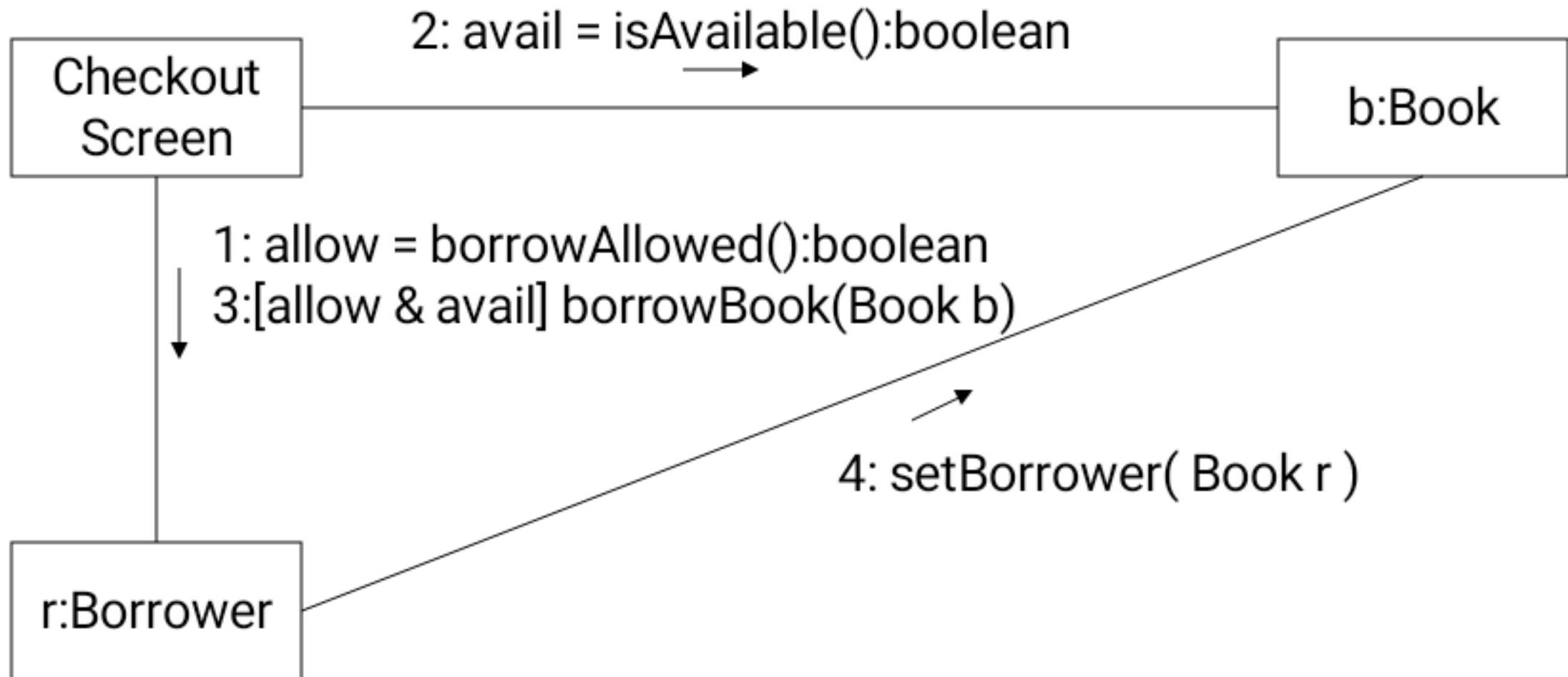
Example: Collaboration Diagram



Interaction (Collaboration) Diagram Notation

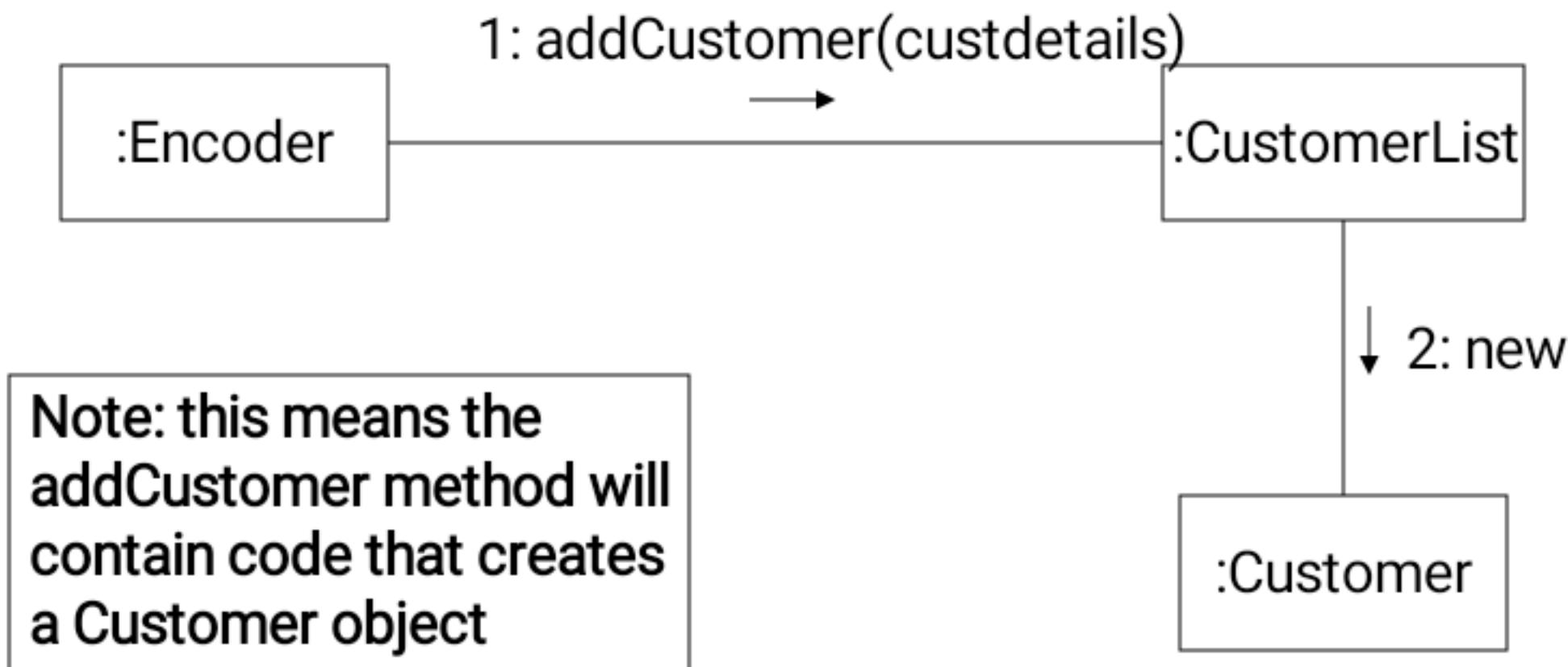
- Rectangles: Classes/Objects
- Arrows: Messages/Method Calls
- Labels on Arrows
 - sequence number (whole numbers or X.X.X notation)
 - method name (the message passed)
 - more details, if helpful and necessary (iterators, conditions, parameters, types, return types)

Including object names, conditions and Types



Creating an Object

- **new** means a constructor is being called
 - Implies object creation

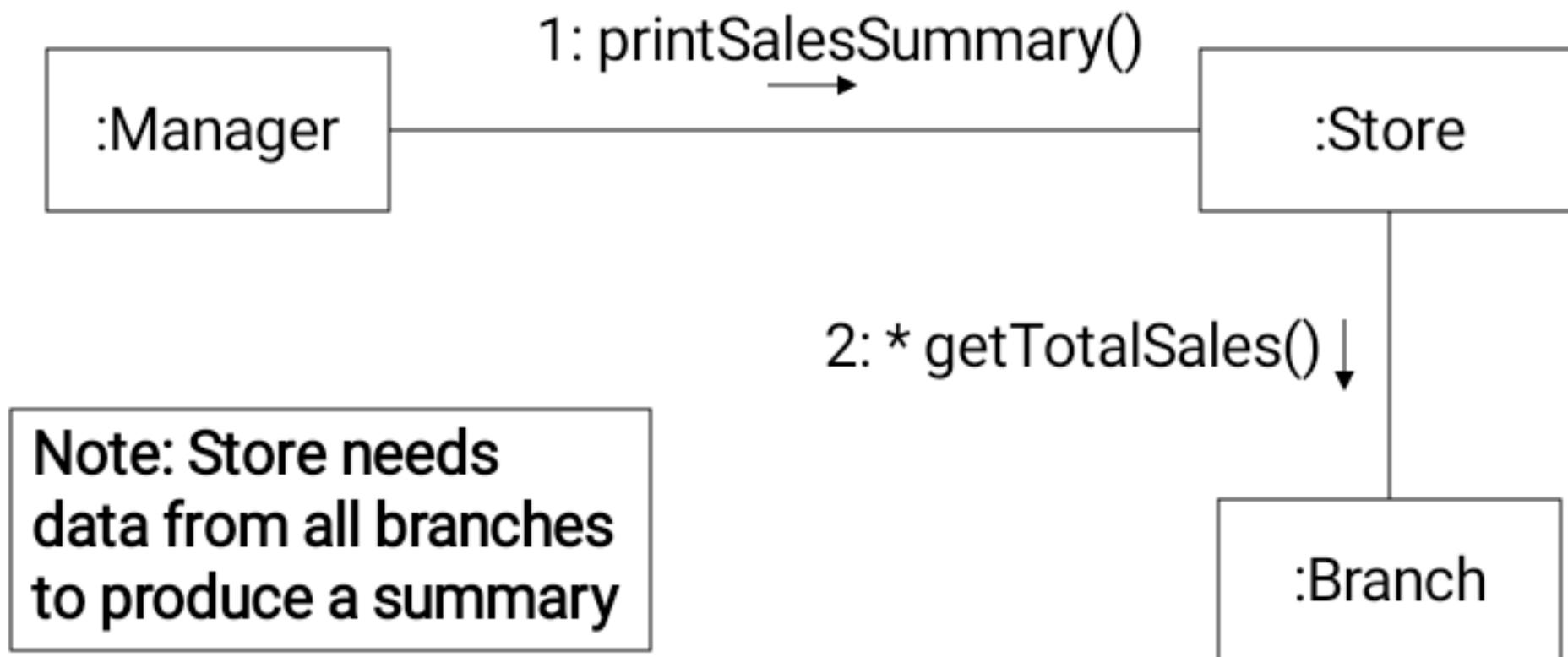


Objects

- Objects participating in a collaboration come in two flavors—supplier and client
- Supplier objects are the objects that supply the method that is being called, and therefore **receive** the message
- Client objects call methods on supplier objects, and therefore **send** messages

Iteration

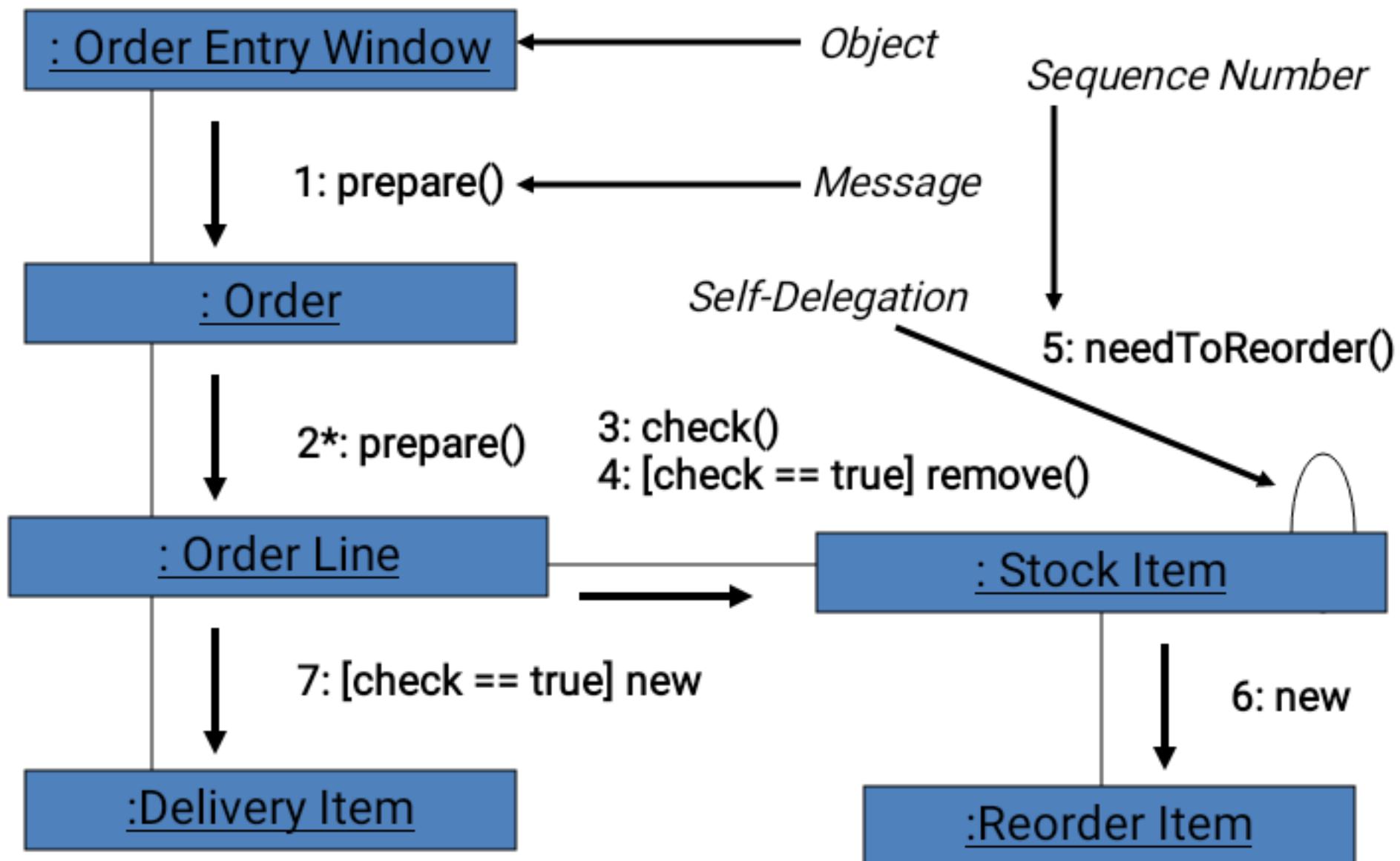
- * is an iterator
 - means the method is called repeatedly



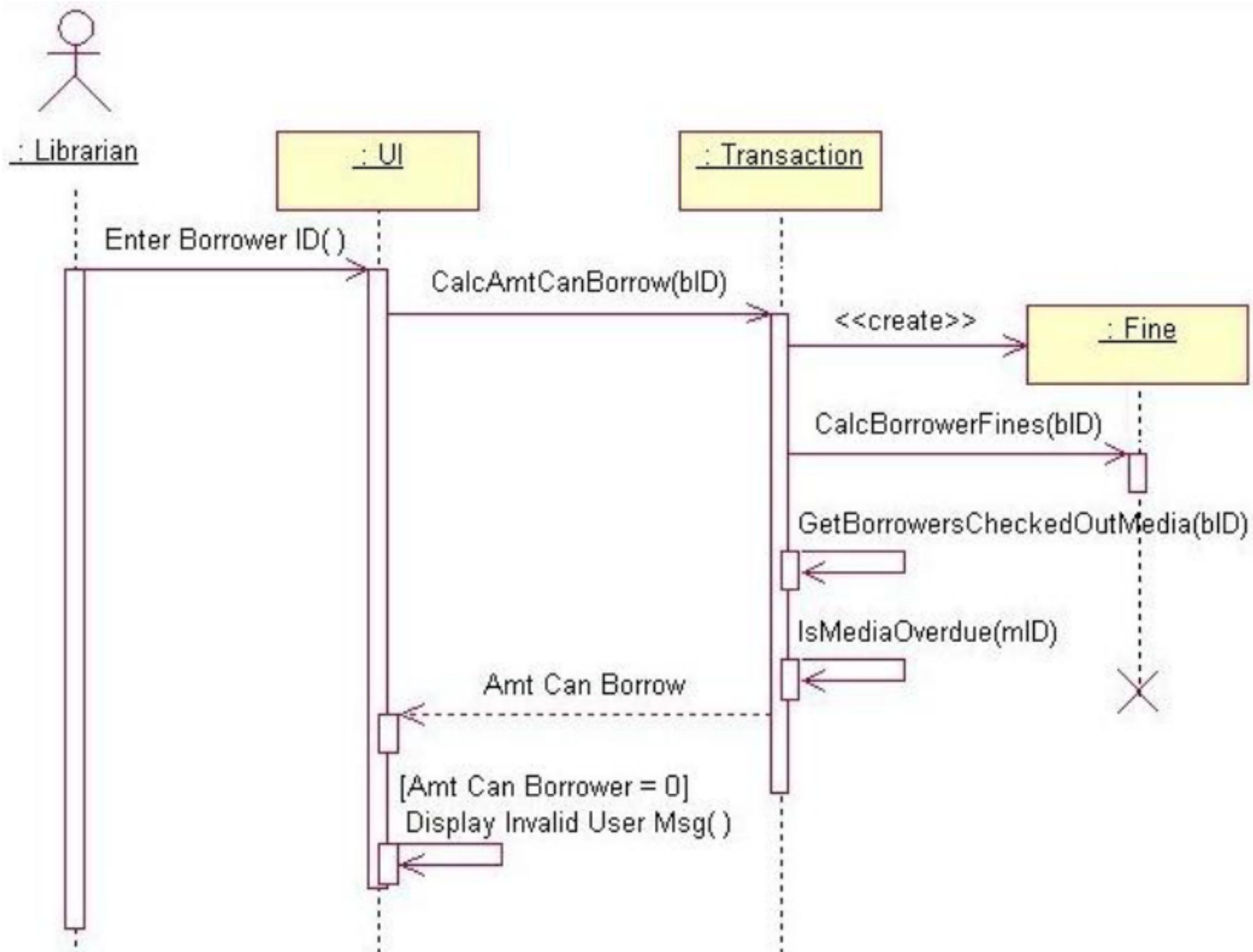
Conditional Messages

- To indicate that a message is run conditionally, prefix the message sequence number with a conditional [guard] clause in brackets [x = true]:
[IsMediaOverdue]
- This indicates that the message is sent only if the condition is met

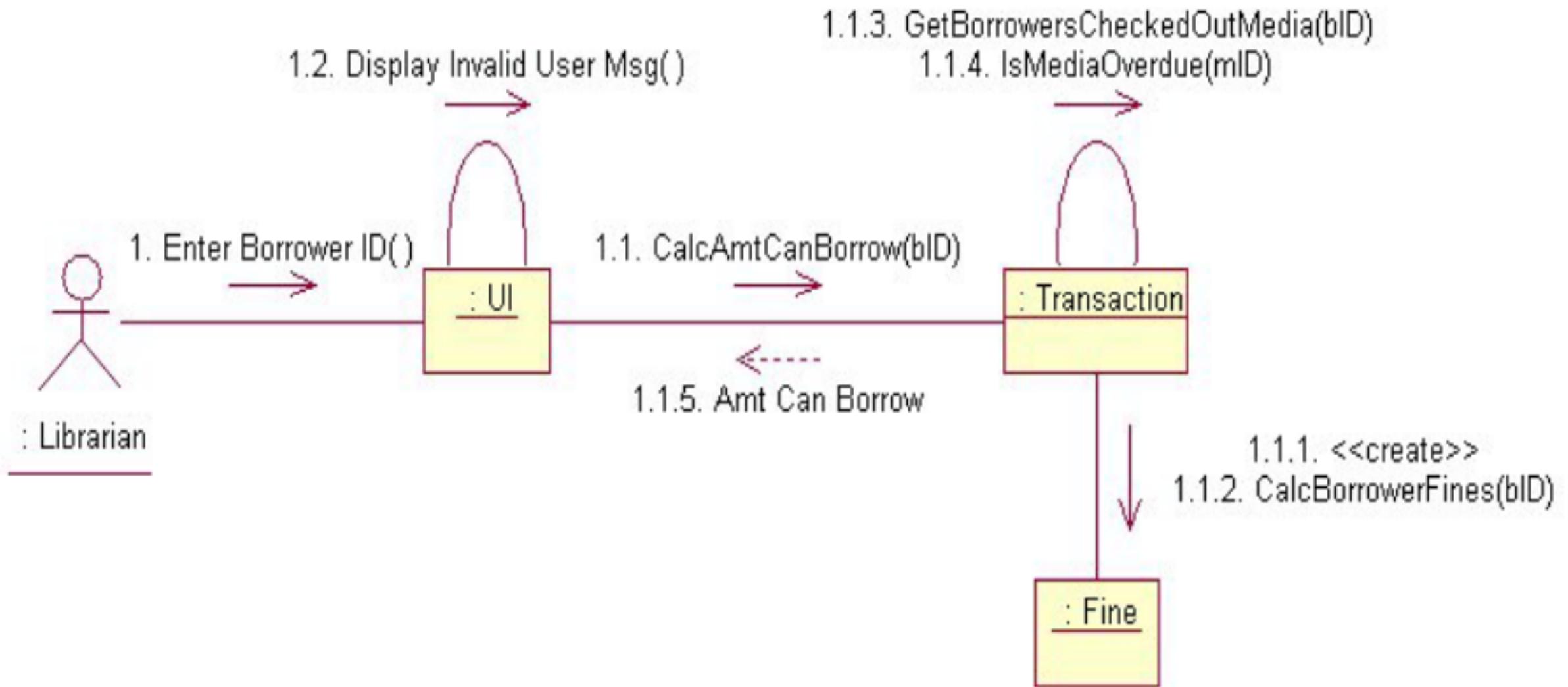
Collaboration Diagram



[Fowler,97]



Sequence diagram is better at 'time ordering'



Collaboration diagram is better at showing the relationship between objects

Collaboration versus Sequence Diagrams

- Use collaboration diagrams when you want to make use of a two-dimensional layout of interacting objects
 - Ok when there aren't that many objects
- Use sequence diagrams when layout doesn't help in presentation and when you want to clarify calling sequence

Collaboration Diagram Sequence Diagram

Both a collaboration diagram and a sequence diagram derive from the same information in the UML's metamodel, so you can take a diagram in one form and convert it into the other. They are semantically equivalent.

In class exercise

- Draw a sequence diagram for:
The Beauty and the Beast kitchen:
 - Draw a sequence diagram for making a peanut butter and jelly sandwich if the following objects are alive: knife, peanut butter jar (and peanut butter), jelly jar (and jelly), bread, plate. I may or may not want the crusts cut off. Don't forget to open and close things like the jars, and put yourself away, cleanup, etc...

In class exercise

- Draw a sequence diagram for:
 - Getting on a flight. Start at home, check in at the counter, go through security, and end up at the gate. (If you have time during the exercise, get yourself to your seat.)
 - You may get searched in security

In class exercise

- Draw a sequence diagram for:
 - Getting money from an ATM machine
 - Treat each part of the ATM as a class
 - Money dispenser
 - Screen
 - Keypad
 - Bank computer
 - Etc...

References

Example diagrams from: <http://www.ibm.com/developerworks/rational/library/3101.html>

Also see Booch G., The Unified Modeling Language User Guide, ch 19.

[Booch99] Booch, Grady, James Rumbaugh, Ivar Jacobson,

The Unified Modeling Language User Guide, Addison Wesley, 1999

[Rumbaugh99] Rumbaugh, James, Ivar Jacobson, Grady Booch, The Unified

Modeling Language Reference Manual, Addison Wesley, 1999

[Jacobson99] Jacobson, Ivar, Grady Booch, James Rumbaugh, The Unified Software Development Process, Addison Wesley, 1999

[Fowler, 1997] Fowler, Martin, Kendall Scott, UML Distilled

(Applying the Standard Object Modeling Language),

Addison Wesley, 1997.

[Brown99] First draft of these slides were created by James Brown.

<http://www.devx.com/codemag/articles/2002/mayjune/collaborationdiagrams/codemag-2.asp>

Software Design (UML)

Inception

Elicitation

Elaboration

Negotiation

Specification

Validation

Requirements
Management

Negotiation Task

- During negotiation, the software engineer reconciles the conflicts between what the customer wants and what can be achieved given limited business resources
- Requirements are **ranked** (i.e., prioritized) by the customers, users, and other stakeholders
- **Risks** associated with each requirement are identified and analyzed
- Rough guesses of development effort are made and used to assess the impact of each requirement on project cost and delivery time
- Using an **iterative** approach, requirements are eliminated, combined and/or modified so that each party achieves some measure of satisfaction

The Art of Negotiation

- Recognize that it is not competition
- Map out a strategy
- Listen actively
- Focus on the other party's interests
- Don't let it get personal
- Be creative
- Be ready to commit

Inception

Elicitation

Elaboration

Negotiation

Specification

Validation

Requirements
Management



Specification Task

- A specification is the final work product produced by the requirements engineer
- It is normally in the form of a **software requirements specification (SRS)**
- It serves as the foundation for subsequent software engineering activities
- It describes the function and performance of a computer-based system and the constraints that will govern its development
- It formalizes the informational, functional, and behavioral requirements of the proposed software in both a graphical and textual format

An SRS Template

INFO



Software Requirements Specification Template

A software requirements specification (SRS) is a document that is created when a detailed description of all aspects of the software to be built must be specified before the project is to commence. It is important to note that a formal SRS is not always written. In fact, there are many instances in which effort expended on an SRS might be better spent in other software engineering activities. However, when software is to be developed by a third party, when a lack of specification would create severe business issues, or when a system is extremely complex or business critical, an SRS may be justified.

Karl Wiegers [Wie03] of Process Impact Inc. has developed a worthwhile template (available at www.processimpact.com/process_assets/srs_template.doc) that can serve as a guideline for those who must create a complete SRS. A topic outline follows:

Table of Contents

Revision History

1. Introduction

- 1.1 Purpose
- 1.2 Document Conventions
- 1.3 Intended Audience and Reading Suggestions
- 1.4 Project Scope
- 1.5 References

2. Overall Description

- 2.1 Product Perspective

- 2.2 Product Features
- 2.3 User Classes and Characteristics
- 2.4 Operating Environment
- 2.5 Design and Implementation Constraints
- 2.6 User Documentation
- 2.7 Assumptions and Dependencies

3. System Features

- 3.1 System Feature 1
- 3.2 System Feature 2 (and so on)

4. External Interface Requirements

- 4.1 User Interfaces
- 4.2 Hardware Interfaces
- 4.3 Software Interfaces
- 4.4 Communications Interfaces

5. Other Nonfunctional Requirements

- 5.1 Performance Requirements
- 5.2 Safety Requirements
- 5.3 Security Requirements
- 5.4 Software Quality Attributes

6. Other Requirements

Appendix A: Glossary

Appendix B: Analysis Models

Appendix C: Issues List

A detailed description of each SRS topic can be obtained by downloading the SRS template at the URL noted earlier in this sidebar.

Typical Contents of a Software Requirements Specification

- Requirements
 - Required states and modes
 - Software requirements grouped by capabilities (i.e., functions, objects)
 - Software external interface requirements
 - Software internal interface requirements
 - Software internal data requirements
 - Other software requirements (safety, security, privacy, environment, hardware, software, communications, quality, personnel, training, logistics, etc.)
 - Design and implementation constraints
- Qualification provisions to ensure each requirement has been met
 - Demonstration, test, analysis, inspection, etc.
- Requirements traceability
 - Trace back to the system or subsystem where each requirement applies

IS IT MORE IMPORTANT
TO FOLLOW OUR DOCU-
MENTED PROCESS OR TO
MEET THE DEADLINE?



www.dilbert.com scottadams@aol.com

I ONLY ASK BECAUSE
OUR DEADLINE IS
ARBITRARY AND OUR
DOCUMENTED PROCESS
WAS PULLED OUT OF
SOMEONE'S LOWER
TORSO.

7-25-06 © 2006 Scott Adams, Inc./Dist. by UFS, Inc.

WHERE'S
YOUR
ARTIFICIAL
SENSE OF
URGENCY?

TEAM-
WORK
KILLED
IT.

SRS sample discussion

SRS for Airline Flight Booking System

Inception

Elicitation

Elaboration

Negotiation

Specification

Validation

Requirements
Management

Validation Task

- During validation, the work products produced as a result of requirements engineering are assessed for quality
- The specification is examined to ensure that
 - all software requirements have been stated unambiguously
 - inconsistencies, omissions, and errors have been detected and corrected
 - the work products conform to the standards established for the process, the project, and the product
- The formal technical review serves as the primary requirements validation mechanism
 - Members include software engineers, customers, users, and other stakeholders

Questions to ask when Validating Requirements

- Is each requirement consistent with the overall objective for the system/product?
- Have all requirements been specified at the proper level of abstraction? That is, do some requirements provide a level of technical detail that is inappropriate at this stage?
- Is the requirement really necessary or does it represent an add-on feature that may not be essential to the objective of the system?
- Is each requirement bounded and unambiguous?
- Does each requirement have attribution? That is, is a source (generally, a specific individual) noted for each requirement?

(more on next slide)

Questions to ask when Validating Requirements (continued)

- Do any requirements conflict with other requirements?
- Is each requirement achievable in the technical environment that will house the system or product?
- Is each requirement testable, once implemented?
 - Approaches: Demonstration, actual test, analysis, or inspection
- Does the requirements model properly reflect the information, function, and behavior of the system to be built?
- Has the requirements model been “partitioned” in a way that exposes progressively more detailed information about the system?

Inception

Elicitation

Elaboration

Negotiation

Specification

Validation

Requirements
Management

Requirements Management Task

- During requirements management, the project team performs a set of activities to identify, control, and track requirements and changes to the requirements at any time as the project proceeds
- Each requirement is assigned a unique identifier
- The requirements are then placed into one or more **traceability tables**
- These tables may be stored in a database that relate features, sources, dependencies, subsystems, and interfaces to the requirements
- A requirements traceability table is also placed at the end of the software requirements specification

References

- Few slides adapted from Dr. Stephen Clyde and Dan Fleck
- <http://www.rational.com/uml/resources.html>

Analysis Modeling

- Requirements analysis
- Flow-oriented modeling
- Scenario-based modeling
- Class-based modeling
- Behavioral modeling

Goals of Analysis Modeling

- Provides the first technical representation of a system
- Is easy to understand and maintain
- Deals with the problem of size by partitioning the system
- Uses graphics whenever possible
- Differentiates between essential information versus implementation information
- Helps in the tracking and evaluation of interfaces
- Provides tools other than narrative text to describe software logic and policy

A Set of Models

- **Flow-oriented modeling** – provides an indication of how data objects are transformed by a set of processing functions
- **Scenario-based modeling** – represents the system from the user's point of view
- **Class-based modeling** – defines objects, attributes, and relationships
- **Behavioral modeling** – depicts the states of the classes and the impact of events on these states

Requirements Analysis

Architectural Design

Slides adapted from various web sources with grateful acknowledgement of the many others who made their course materials freely available online.

Introduction

Definitions

- Design is an activity concerned with major decisions, often of a structural nature.
- The software architecture of a program or computing system is the structure or structures of the system which comprise – [Bass, Clements, and Kazman]
 - The software components
 - The externally visible properties of those components
 - The relationships among the components
- Software architectural design represents the structure of the data and program components that are required to build a computer-based system
- An architectural design model is transferable
 - It can be applied to the design of other systems
 - It represents a set of abstractions that enable software engineers to describe architecture in predictable ways

Architectural Design Process

- Basic Steps
 - Creation of the data design
 - Derivation of one or more representations of the architectural structure of the system
 - Analysis of alternative architectural styles to choose the one best suited to customer requirements and quality attributes
 - Elaboration of the architecture based on the selected architectural style
- A database designer creates the data architecture for a system to represent the data components
- A system architect selects an appropriate architectural style derived during system engineering and software requirements analysis

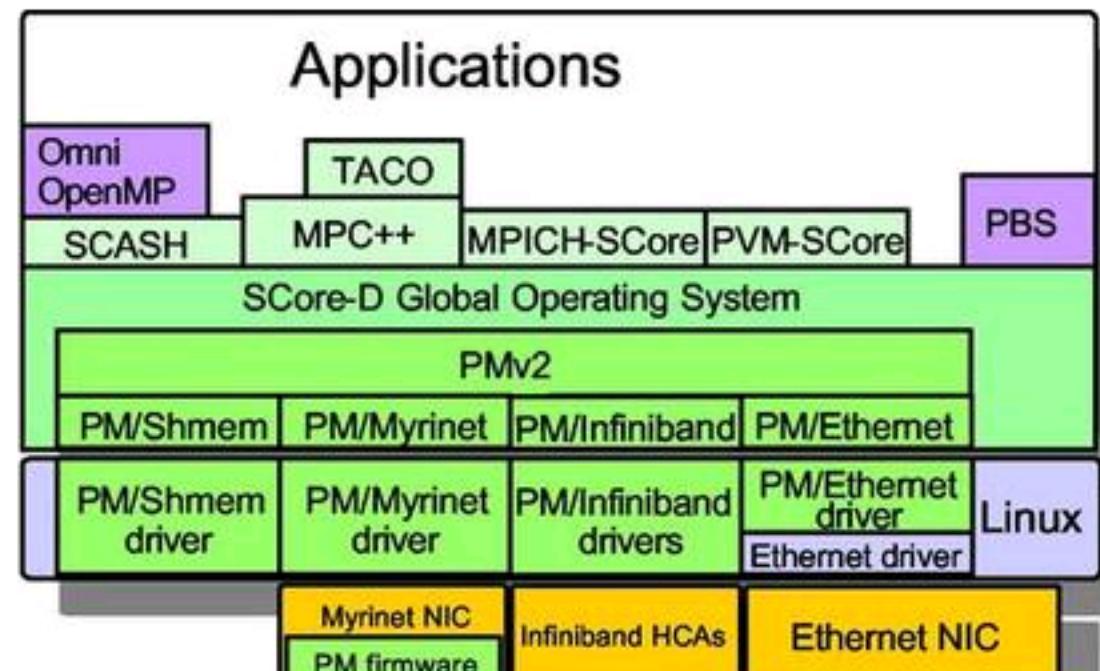
Emphasis on Software Components

- A software architecture enables a software engineer to
 - Analyze the effectiveness of the design in meeting its stated requirements
 - Consider architectural alternatives at a stage when making design changes is still relatively easy
 - Reduce the risks associated with the construction of the software
- Focus is placed on the software component
 - A program module
 - An object-oriented class
 - A database
 - Middleware

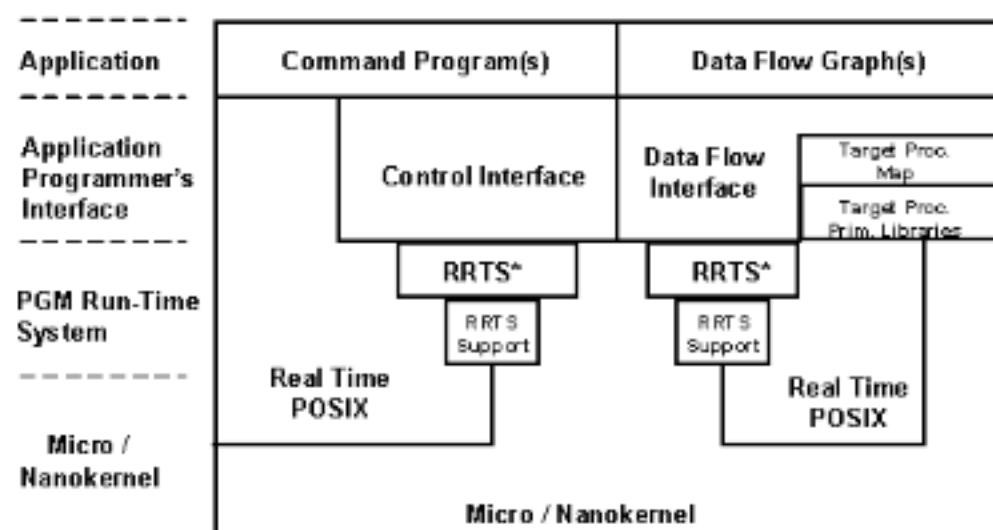
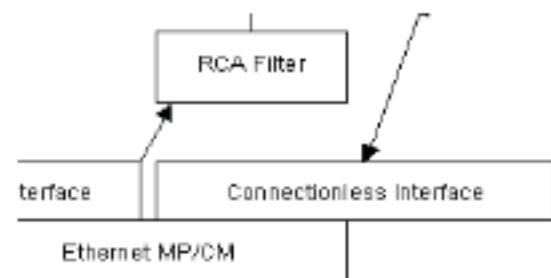
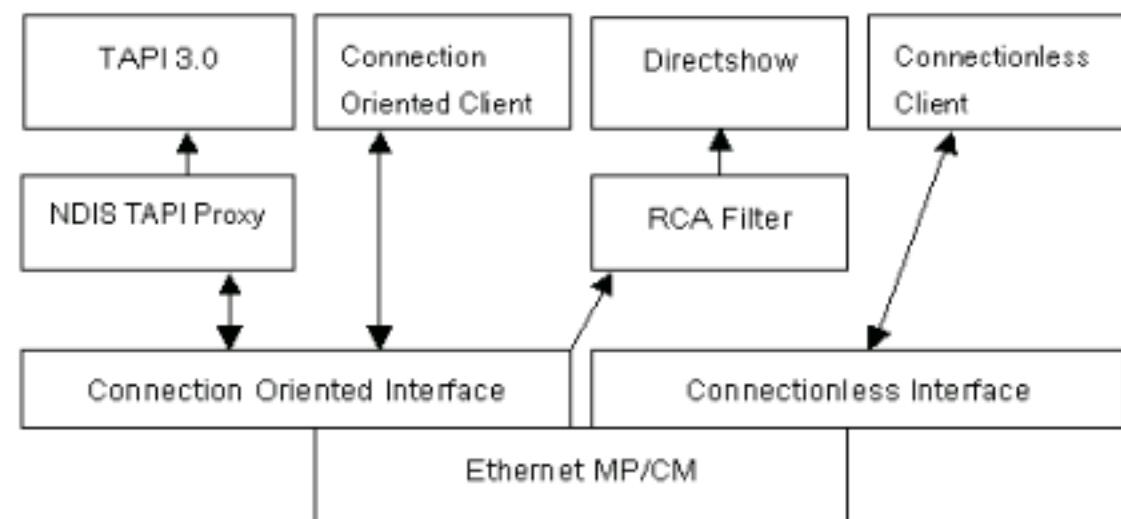
Importance of Software Architecture [Bass]

- Representations of software architecture are an enabler for communication between all stakeholders interested in the development of a computer-based system
- The software architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity
- The software architecture constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together

Example Software Architecture Diagrams



Two types of Infiniband drivers are available: for Fujitsu and TopSPIN



*RRTS: RASSP Run-Time Support

Data Design

Purpose of Data Design

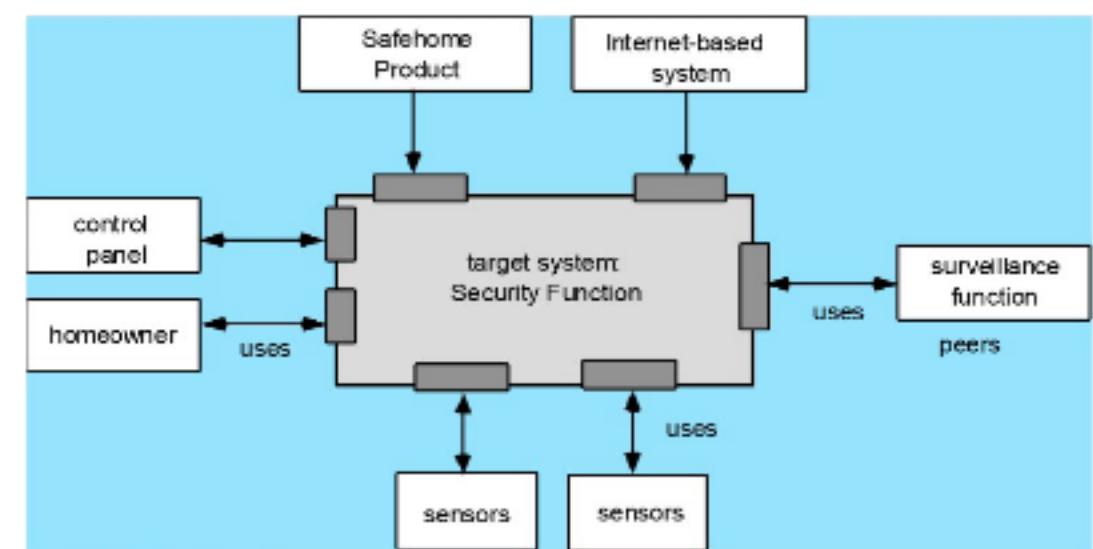
- Data design translates data objects defined as part of the analysis model into
 - Data structures at the software component level
 - A possible database architecture at the application level
- It focuses on the representation of data structures that are directly accessed by one or more software components
- The challenge is to store and retrieve the data in such way that useful information can be extracted from the data environment
- "Data quality is the difference between a data warehouse and a data garbage dump"

Data Design Principles

- The systematic analysis principles that are applied to function and behavior should also be applied to data
- All data structures and the operations to be performed on each one should be identified
- A mechanism for defining the content of each data object should be established and used to define both data and the operations applied to it
- Low-level data design decisions should be deferred until late in the design process
- The representation of a data structure should be known only to those modules that must make direct use of the data contained within the structure
- A library of useful data structures and the operations that may be applied to them should be developed
- A software programming language should support the specification and realization of abstract data types

Software Architectural Styles

Common Architectural Styles of Homes



Common Architectural Styles of Homes

A-Frame

Four square

Ranch

Bungalow

Georgian

Split level

Cape Cod

Greek Revival

Tidewater

Colonial

Prairie Style

Tudor

Federal

Pueblo

Victorian

Software Architectural Style

- The software that is built for computer-based systems exhibit one of many architectural styles
- Each style describes a system category that encompasses
 - A set of component types that perform a function required by the system
 - A set of connectors (subroutine call, remote procedure call, data stream, socket) that enable communication, coordination, and cooperation among components
 - Semantic constraints that define how components can be integrated to form the system
 - A topological layout of the components indicating their runtime interrelationships

A Taxonomy of Architectural Styles

Independent Components

Communicating
Processes

Client/Server

Peer-to-Peer

Event Systems

Implicit
Invocation

Explicit
Invocation

Data Flow

Batch Sequential

Pipe and
Filter

Data-Centered

Repository

Blackboard

Virtual Machine

Interpreter

Rule-Based
System

Call and Return

Main Program
and Subroutine

Layered

Object
Oriented

Remote Procedure Call

Data Flow Style

- Has the goal of modifiability
- Characterized by viewing the system as a series of transformations on successive pieces of input data
- Data enters the system and then flows through the components one at a time until they are assigned to output or a data store
- Batch sequential style
 - The processing steps are independent components
 - Each step runs to completion before the next step begins
- Pipe-and-filter style
 - Emphasizes the incremental transformation of data by successive components
 - The filters incrementally transform the data (entering and exiting via streams)
 - The filters use little contextual information and retain no state between instantiations
 - The pipes are ~~stateless and exist only~~ exist to move data between filters

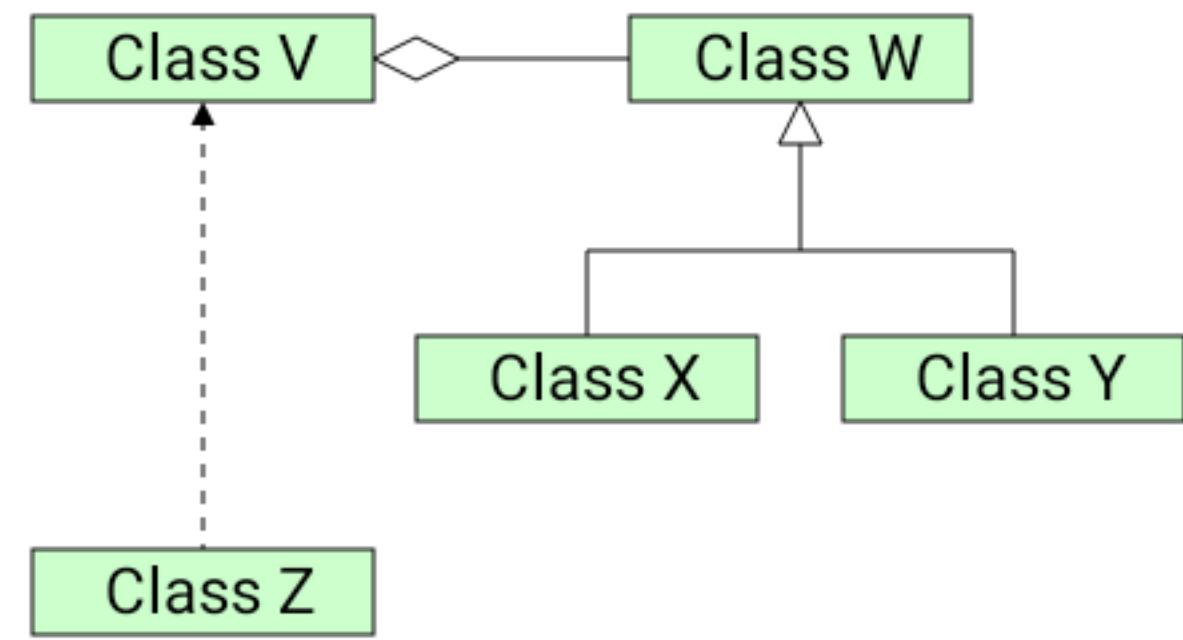
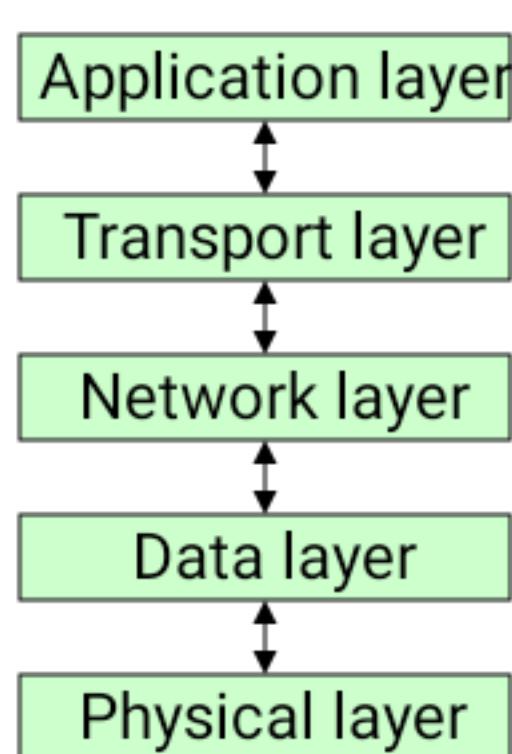
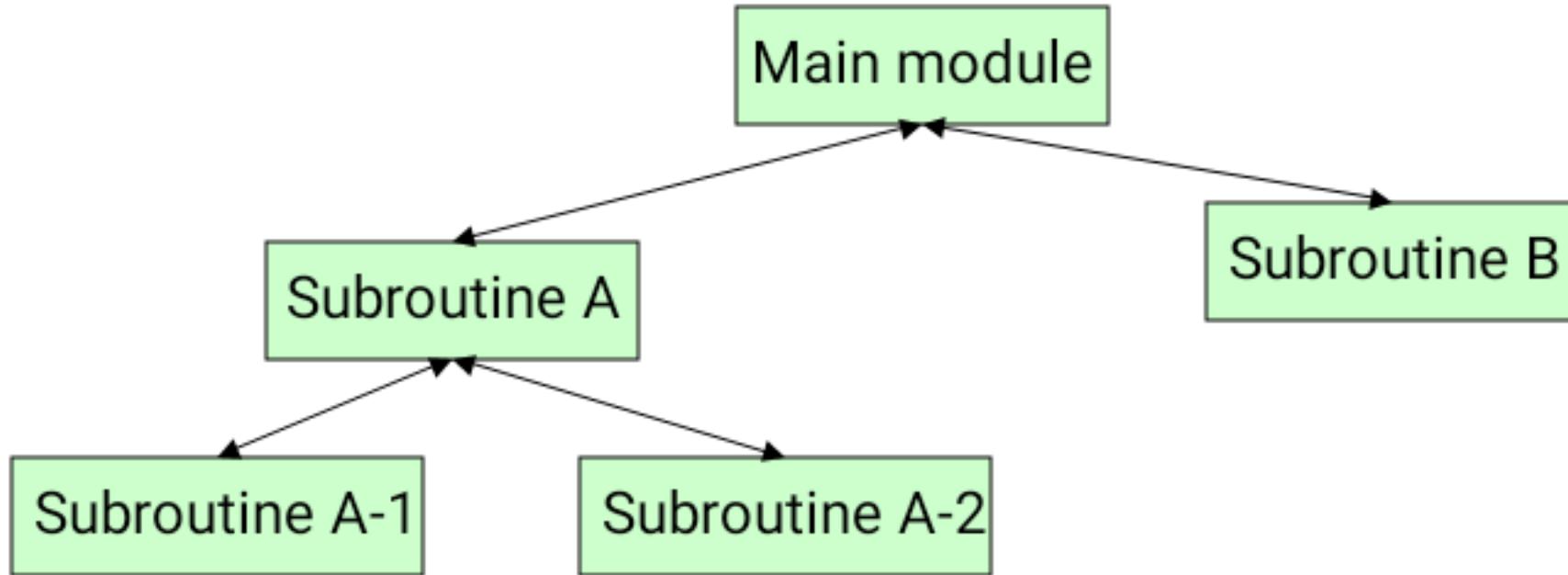
Data Flow Style (continued)

- Advantages
 - Has a simplistic design in the limited ways in which the components interact with the environment
 - Consists of no more and no less than the construction of its parts
 - Simplifies reuse and maintenance
 - Is easily made into a parallel or distributed execution in order to enhance system performance
- Disadvantages
 - Implicitly encourages a batch mentality so interactive applications are difficult to create in this style
 - Ordering of filters can be difficult to maintain so the filters cannot cooperatively interact to solve a problem
 - Exhibits poor performance
 - Filters typically force the least common denominator of data representation (usually ASCII stream)
 - Filter may need unlimited buffers if they cannot start producing output until they receive all of the input
 - Each filter operates as a separate process or procedure call, thus incurring overhead in set-up and take-down time
(More on next slide)

Data Flow Style (continued)

- Use this style when it makes sense to view your system as one that produces a well-defined easily identified output
 - The output should be a direct result of sequentially transforming a well-defined easily identified input in a time-independent fashion

Call-and-Return Style

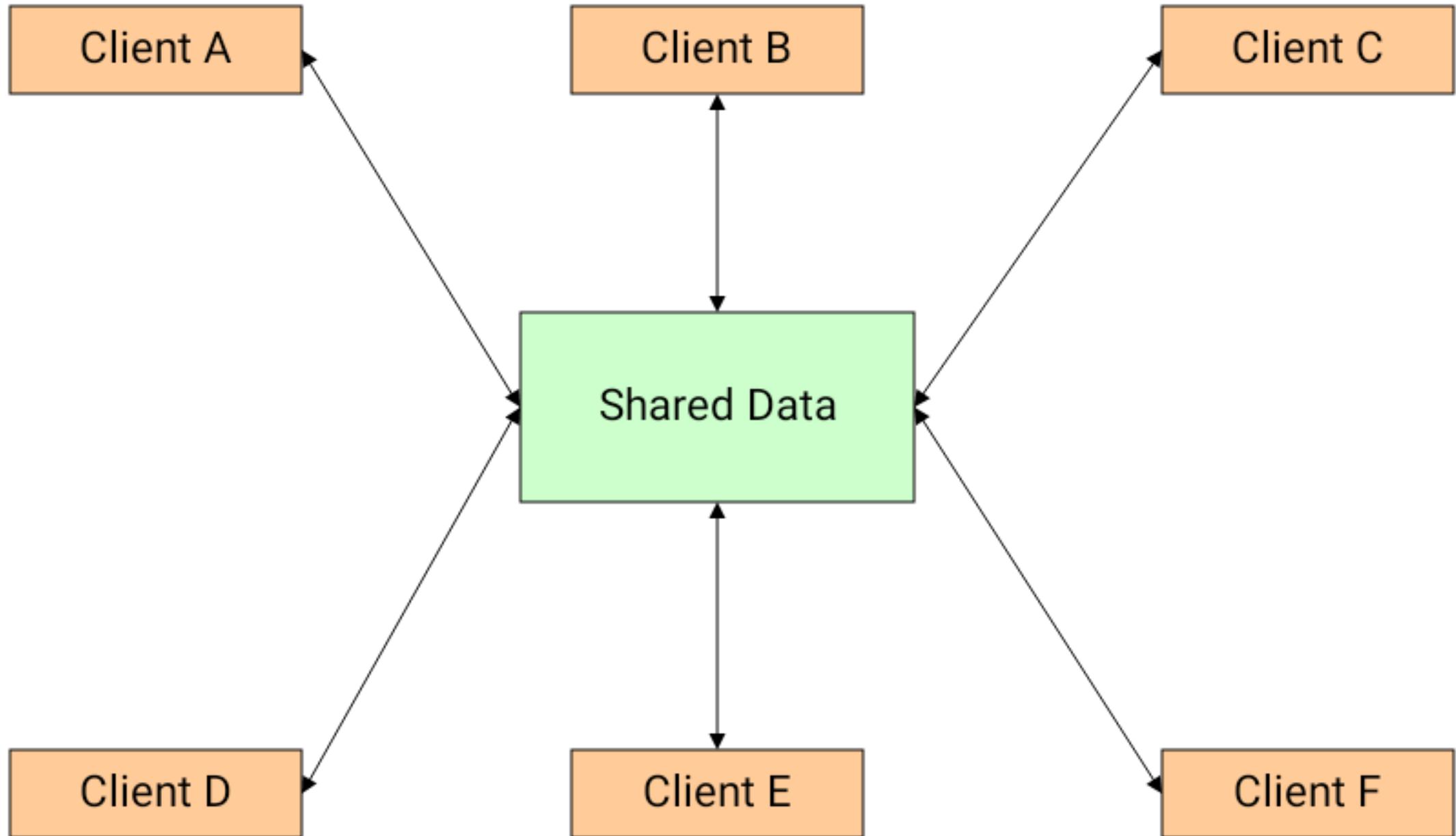


Call-and-Return Style

Call-and-Return Style (continued)

- Object-oriented or abstract data type system
 - Emphasizes the bundling of data and how to manipulate and access data
 - Keeps the internal data representation hidden and allows access to the object only through provided operations
 - Permits inheritance and polymorphism
- Layered system
 - Assigns components to layers in order to control inter-component interaction
 - Only allows a layer to communicate with its immediate neighbor
 - Assigns core functionality such as hardware interfacing or system kernel operations to the lowest layer
 - Builds each successive layer on its predecessor, hiding the lower layer and providing services for the upper layer
 - Is compromised by layer bridging that skips one or more layers to improve runtime performance
- Use this style when the order of computation is fixed, when interfaces are specific, and when components can make no useful progress while awaiting the results of request to other components

Data-Centered Style



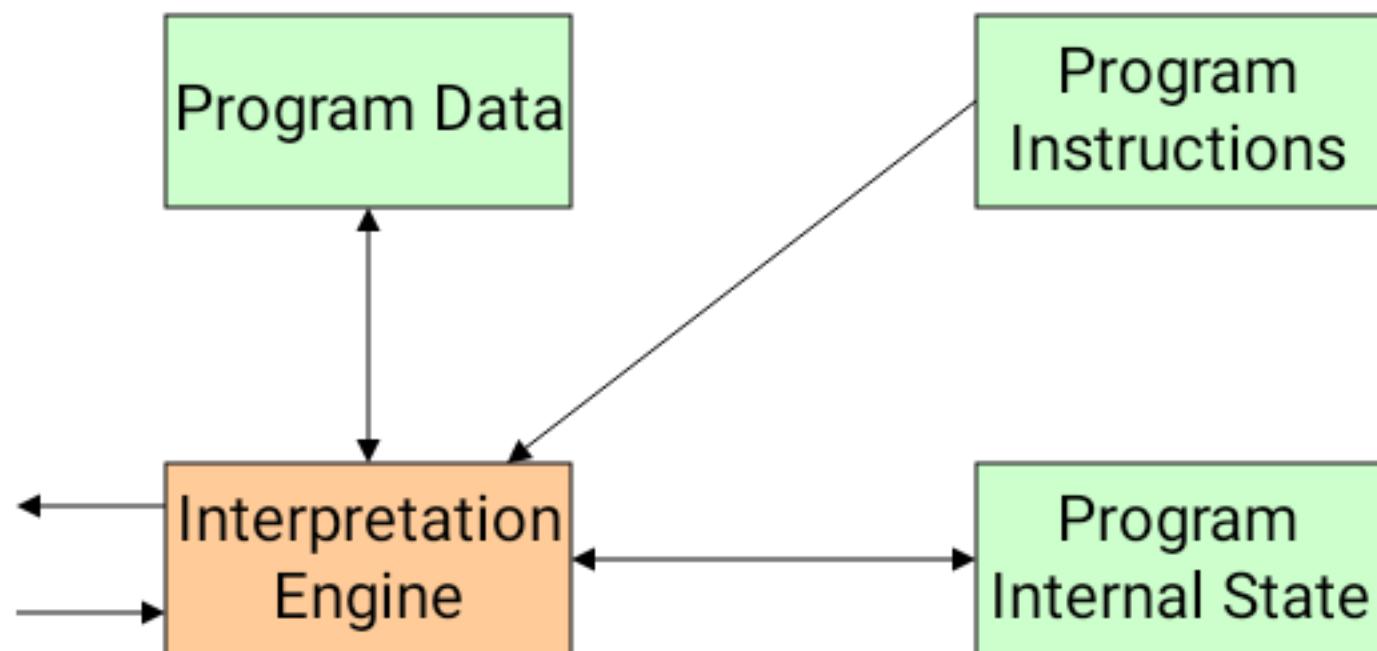
Data-Centered Style (continued)

- Has the goal of integrating the data
- Refers to systems in which the access and update of a widely accessed data store occur
- A client runs on an independent thread of control
- The shared data may be a passive repository or an active blackboard
 - A blackboard notifies subscriber clients when changes occur in data of interest
- At its heart is a centralized data store that communicates with a number of clients
- Clients are relatively independent of each other so they can be added, removed, or changed in functionality
- The data store is independent of the clients

Data-Centered Style (continued)

- Use this style when a central issue is the storage, representation, management, and retrieval of a large amount of related persistent data
- Note that this style becomes client/server if the clients are modeled as independent processes

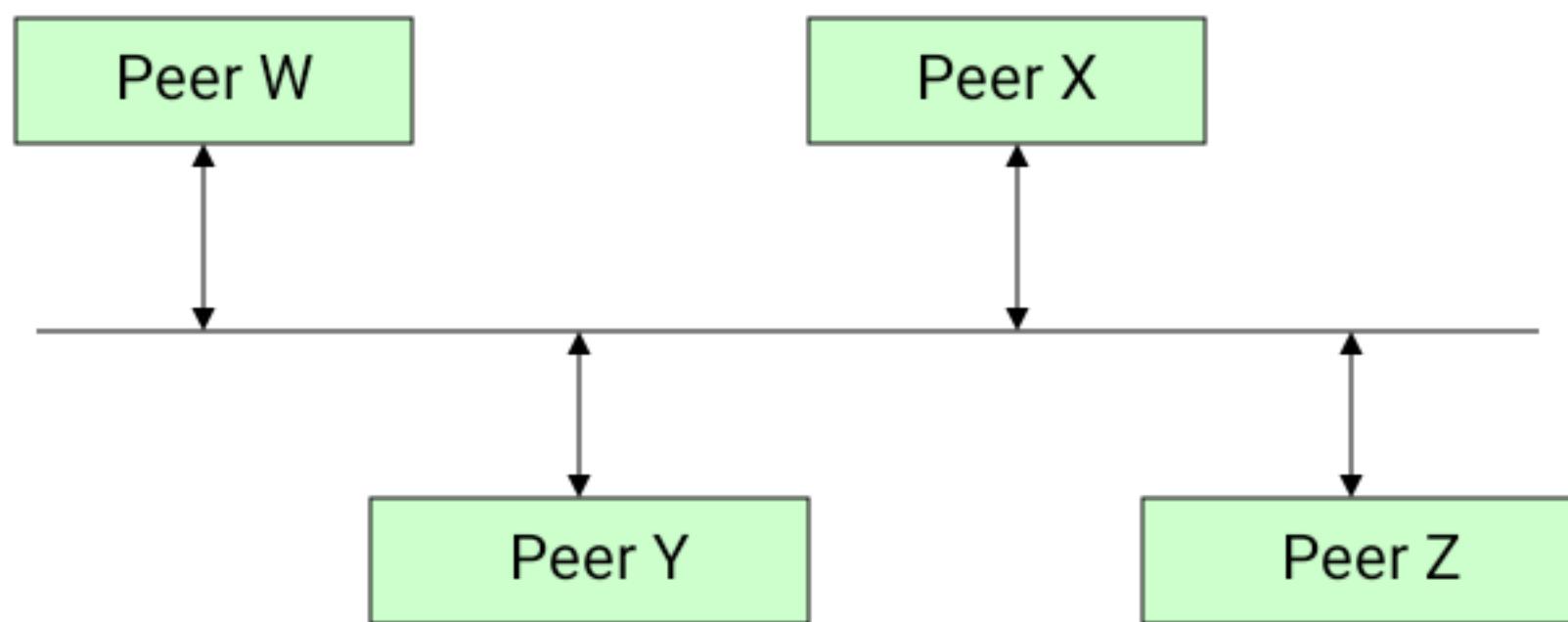
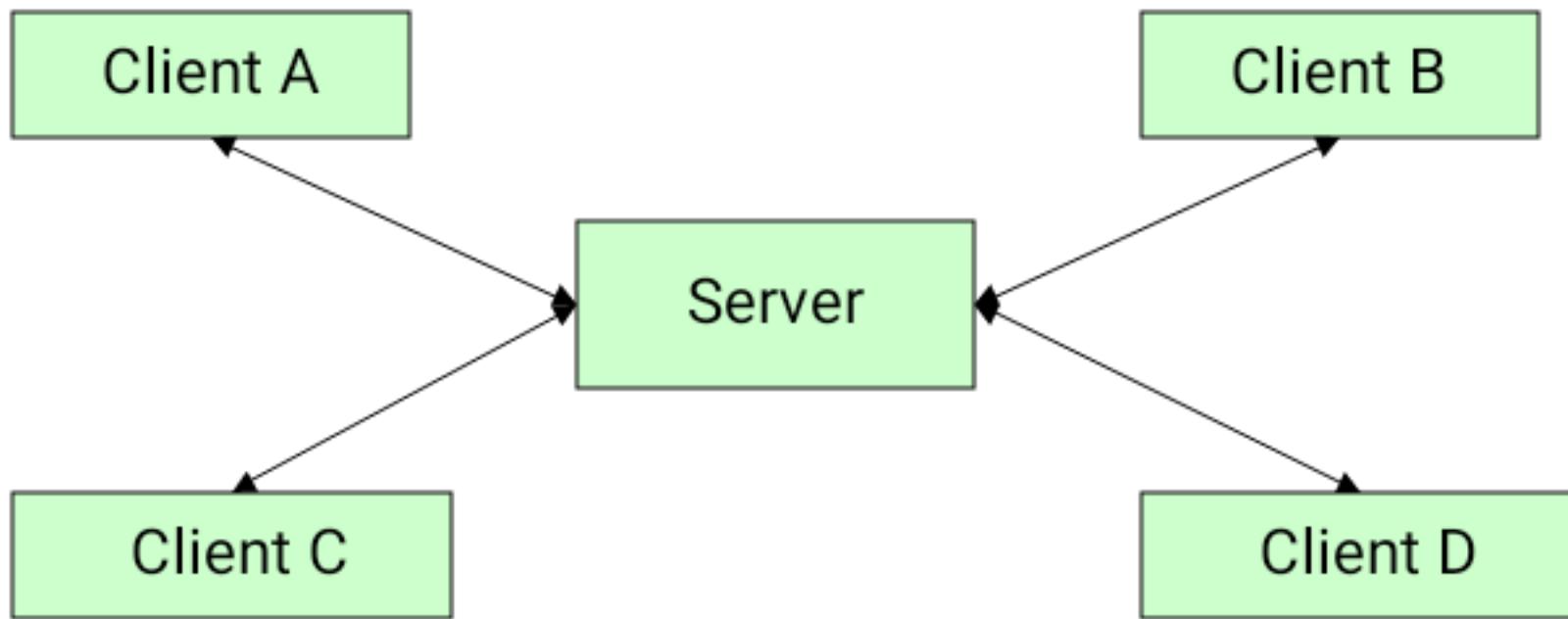
Virtual Machine Style



Virtual Machine Style

- Has the goal of portability
- Software systems in this style simulate some functionality that is not native to the hardware and/or software on which it is implemented
 - Can simulate and test hardware platforms that have not yet been built
 - Can simulate "disaster modes" as in flight simulators or safety-critical systems that would be too complex, costly, or dangerous to test with the real system
- Examples include interpreters, rule-based systems, and command language processors
- Interpreters
 - Add flexibility through the ability to interrupt and query the program and introduce modifications at runtime
 - Incur a performance cost because of the additional computation involved in execution
- Use this style when you have developed a program or some form

Independent Component Style



Independent Component Style

- Consists of a number of independent processes that communicate through messages
- Has the goal of modifiability by decoupling various portions of the computation
- Sends data between processes but the processes do not directly control each other
- Event systems style
 - Individual components announce data that they wish to share (publish) with their environment
 - The other components may register an interest in this class of data (subscribe)
 - Makes use of a message component that manages communication among the other components
 - Components publish information by sending it to the message manager
 - When the data appears, the subscriber is invoked and receives the data
 - Decouples component implementation from knowing the names and locations of other components

(More on next slide)

Independent Component Style (continued)

- Communicating processes style
 - These are classic multi-processing systems
 - Well-known subtypes are client/server and peer-to-peer
 - The goal is to achieve scalability
 - A server exists to provide data and/or services to one or more clients
 - The client originates a call to the server which services the request
- Use this style when
 - Your system has a graphical user interface
 - Your system runs on a multiprocessor platform
 - Your system can be structured as a set of loosely coupled components
 - Performance tuning by reallocating work among processes is important
 - Message passing is sufficient as an interaction mechanism among components

Heterogeneous Styles

- Systems are seldom built from a single architectural style
- Three kinds of heterogeneity
 - Locationally heterogeneous
 - The drawing of the architecture reveals different styles in different areas (e.g., a branch of a call-and-return system may have a shared repository)
 - Hierarchically heterogeneous
 - A component of one style, when decomposed, is structured according to the rules of a different style
 - Simultaneously heterogeneous
 - Two or more architectural styles may both be appropriate descriptions for the style used by a computer-based system

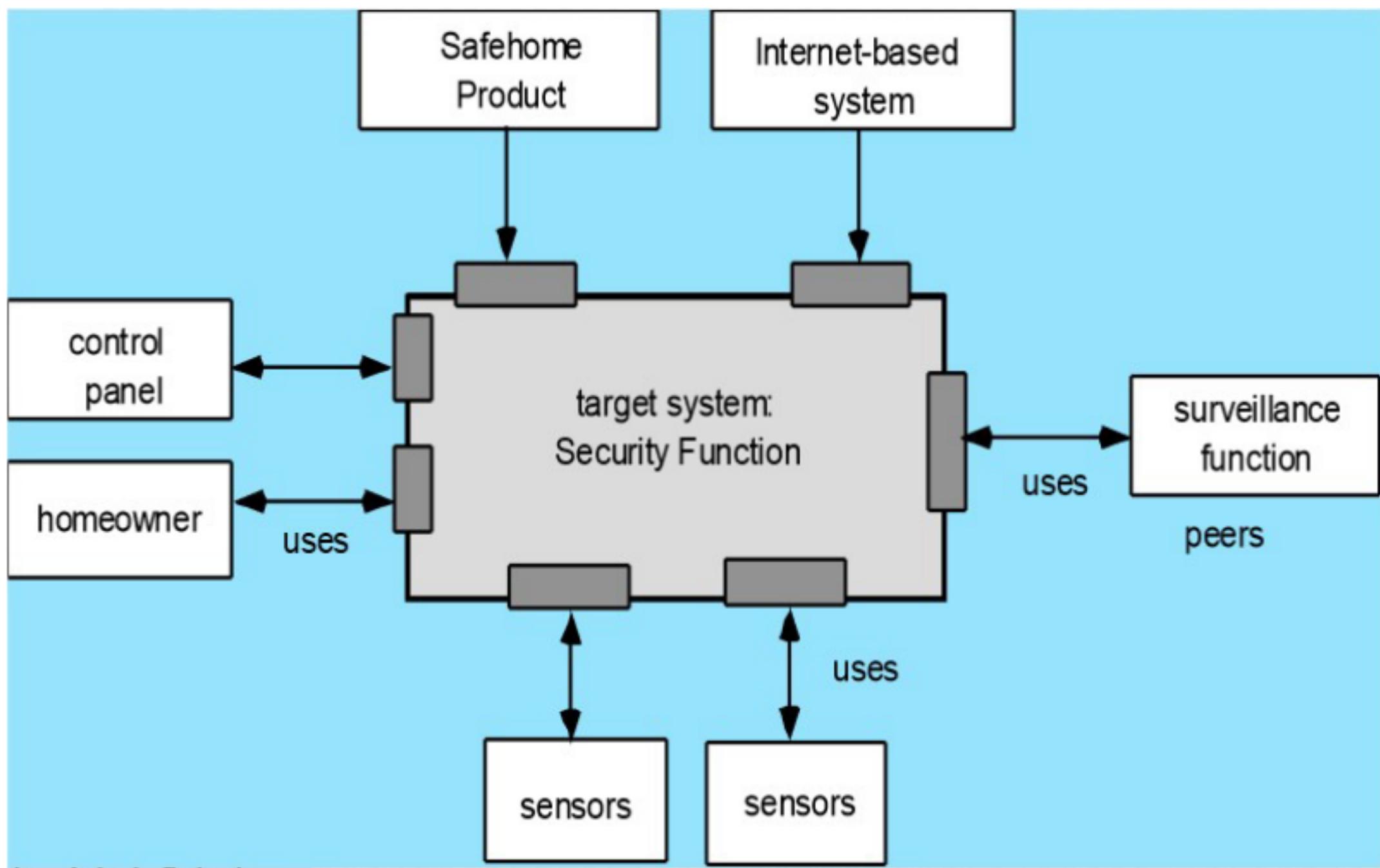
Architectural Design Process

Architectural Design Steps

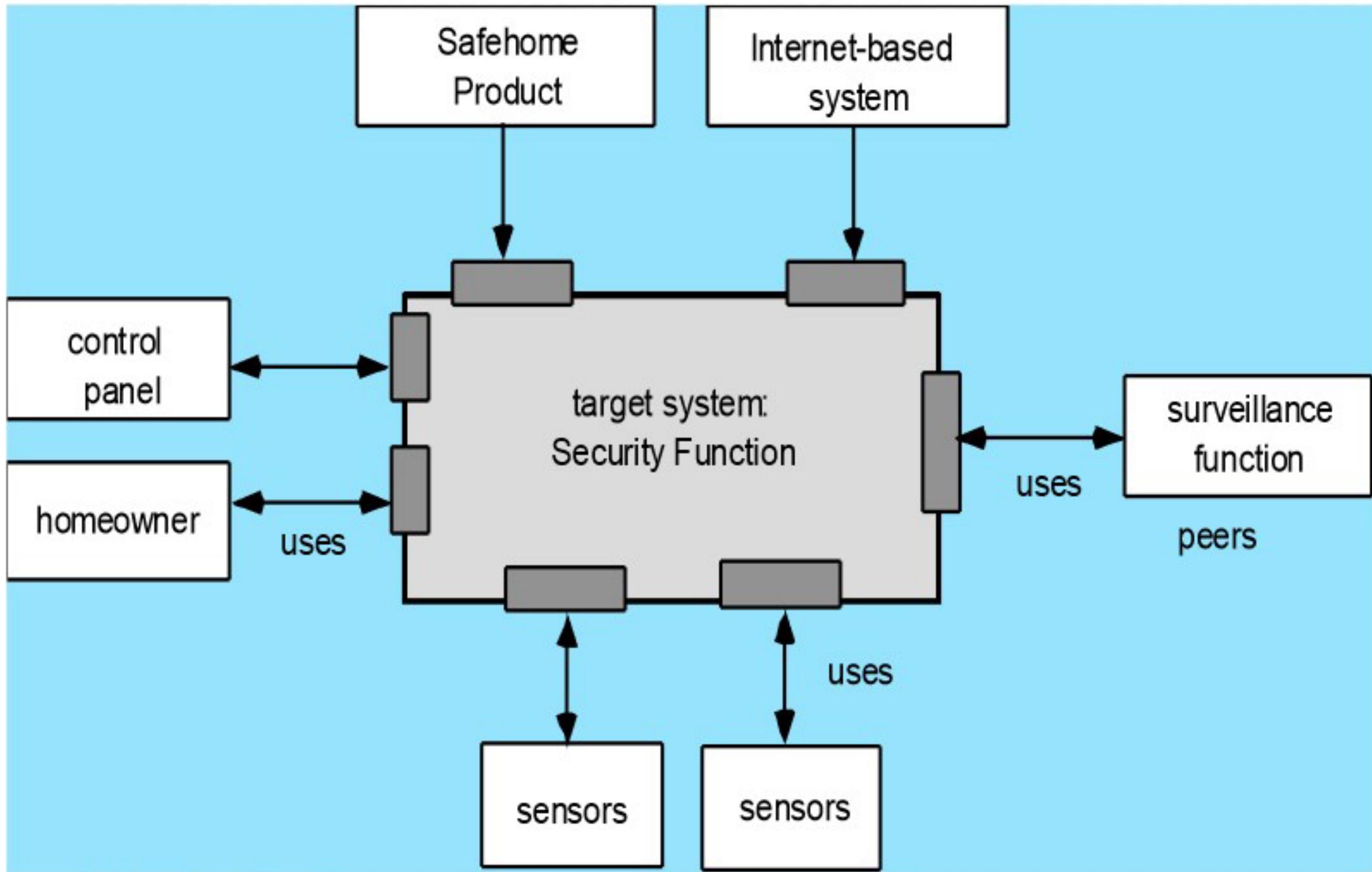
- Represent the system in context
- Define archetypes
- Refine the architecture into components
- Describe instantiations of the system

"A doctor can bury his mistakes, but an architect can only advise his client to plant vines." Frank Lloyd Wright

Architectural context diagram



1. Represent the System in Context



Architectural Context Diagram (ACD)

1. Represent the System in Context (continued)

- Use an **Architectural Context Diagram** (ACD) that shows
 - The identification and flow of all information into and out of a system
 - The specification of all interfaces
 - Any relevant support processing from/by other systems
- An ACD models the manner in which software interacts with entities external to its boundaries
- An ACD identifies systems that interoperate with the target system
 - **Super-ordinate systems**
 - Use target system as part of some higher level processing scheme
 - **Sub-ordinate systems**
 - Used by target system and provide necessary data or processing
 - **Peer-level systems**
 - Interact on a peer-to-peer basis with target system to produce or consume data
 - **Actors**
 - People or devices that interact with target system to produce or consume data

2. Define Archetypes

- Abstract building blocks of an architectural design.
- Archetypes indicate the important abstractions within the problem domain (i.e., they model information)
- An archetype is a class or pattern that represents a core abstraction that is critical to the design of an architecture for the target system
- It is also an abstraction from a class of programs with a common structure and includes class-specific design strategies and a collection of example program designs and implementations
- Only a relatively small set of archetypes is required in order to design even relatively complex systems
- The target system architecture is composed of these archetypes
 - They represent stable elements of the architecture
 - They may be instantiated in different ways based on the behavior of the system
 - They can be derived from the analysis class model
- The archetypes and their relationships can be illustrated in a UML class diagram

Example Archetypes in Humanity

- Addict/Gambler
- Amateur
- Beggar
- Clown
- Companion
- Damsel in distress
- Destroyer
- Detective
- Don Juan
- Drunk
- Engineer
- Father
- Gossip
- Guide
- Healer
- Hero
- Judge
- King
- Knight
- Liberator/Rescuer
- Lover/Devotee
- Martyr
- Mediator
- Mentor/Teacher
- Messiah/Savior
- Monk/Nun
- Mother
- Mystic/Hermit
- Networker
- Pioneer
- Poet
- Priest/Minister
- Prince
- Prostitute
- Queen
- Rebel/Pirate
- Saboteur
- Samaritan
- Scribe/Journalist
- Seeker/Wanderer
- Servant/Slave
- Storyteller
- Student
- Trickster-Thief
- Vampire
- Victim
- Virgin
- Visionary/Prophet
- Warrior/Soldier

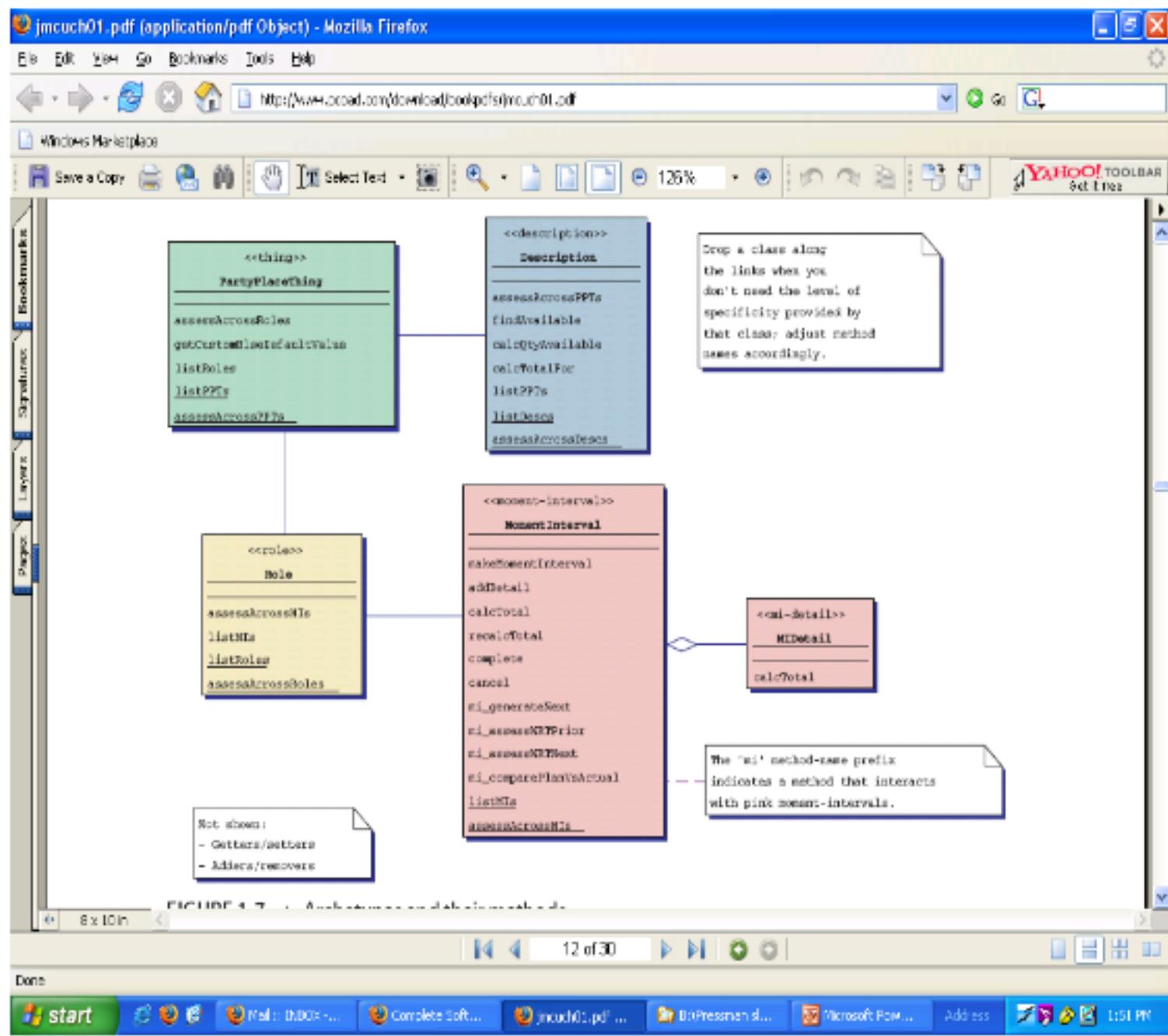
Example Archetypes in Software Architecture

- Node
- Detector/Sensor
- Indicator
- Controller
- Manager
- Moment-Interval
- Role
- Description
- Party, Place, or Thing

(Source: Pressman)

(Source: Archetypes, Color, and the Domain Neutral Component)

UML relationships for *SafeHome* security function archetypes



Archetypes – their attributes

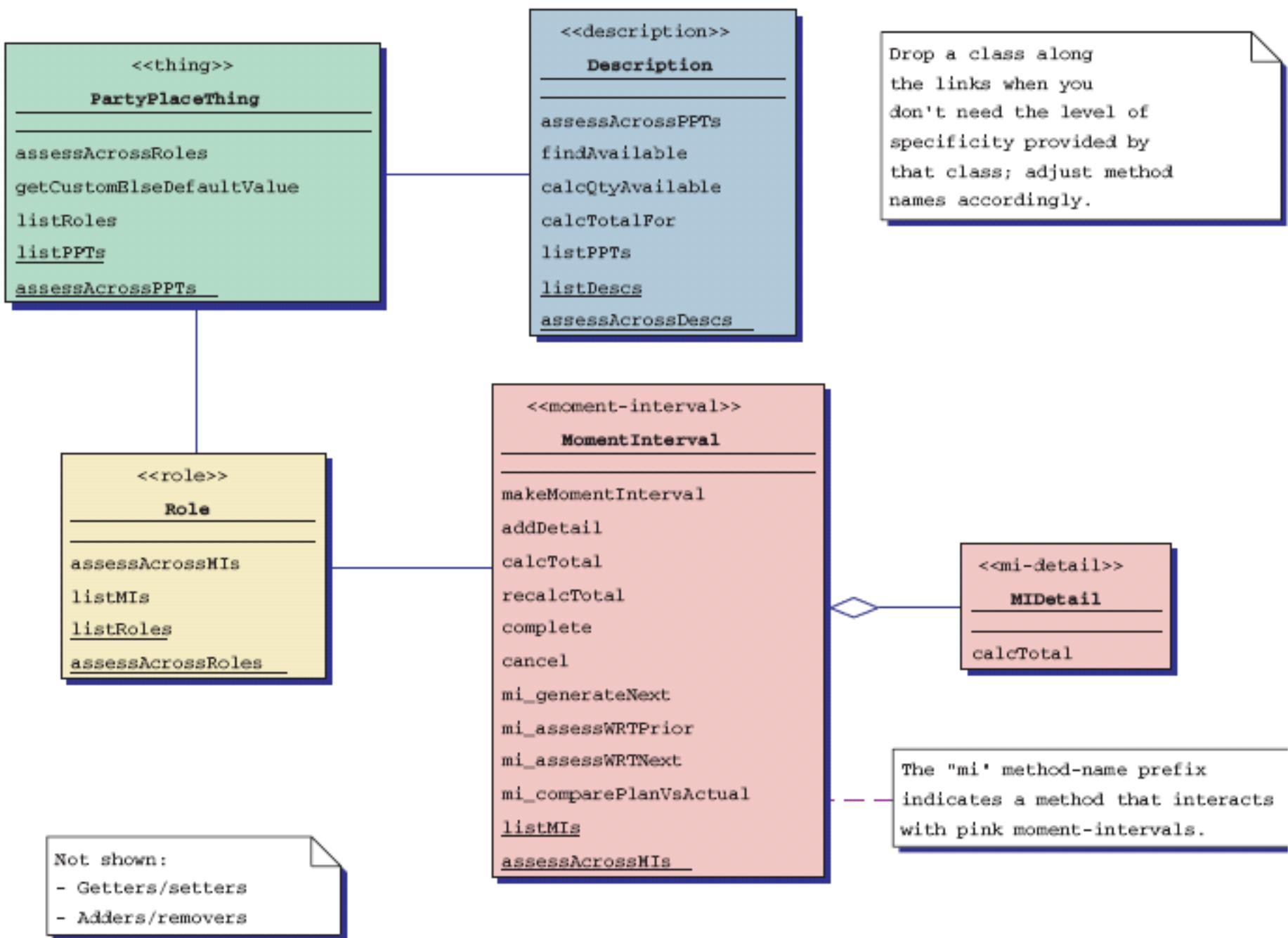
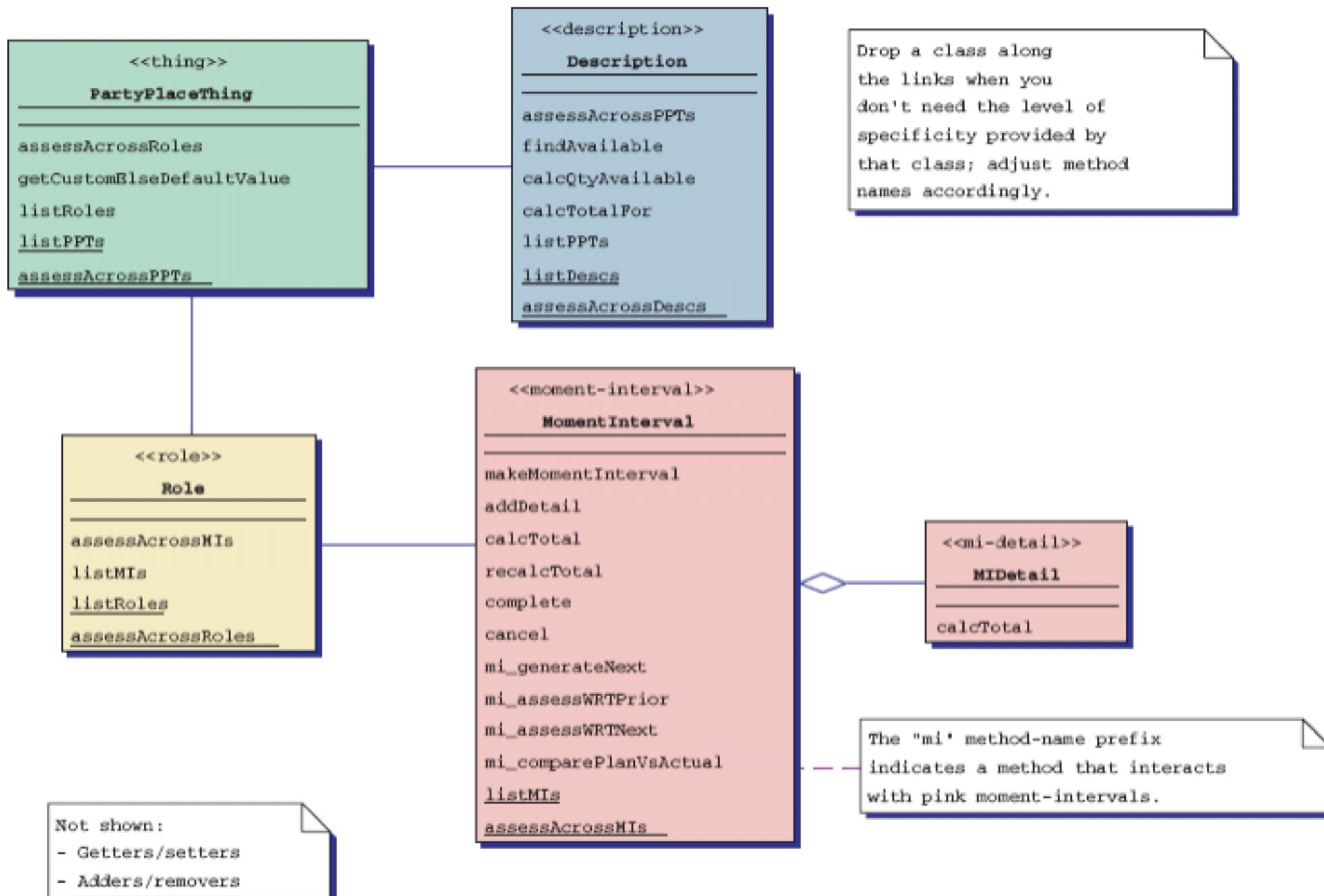


FIGURE 4.7 – Archetypes

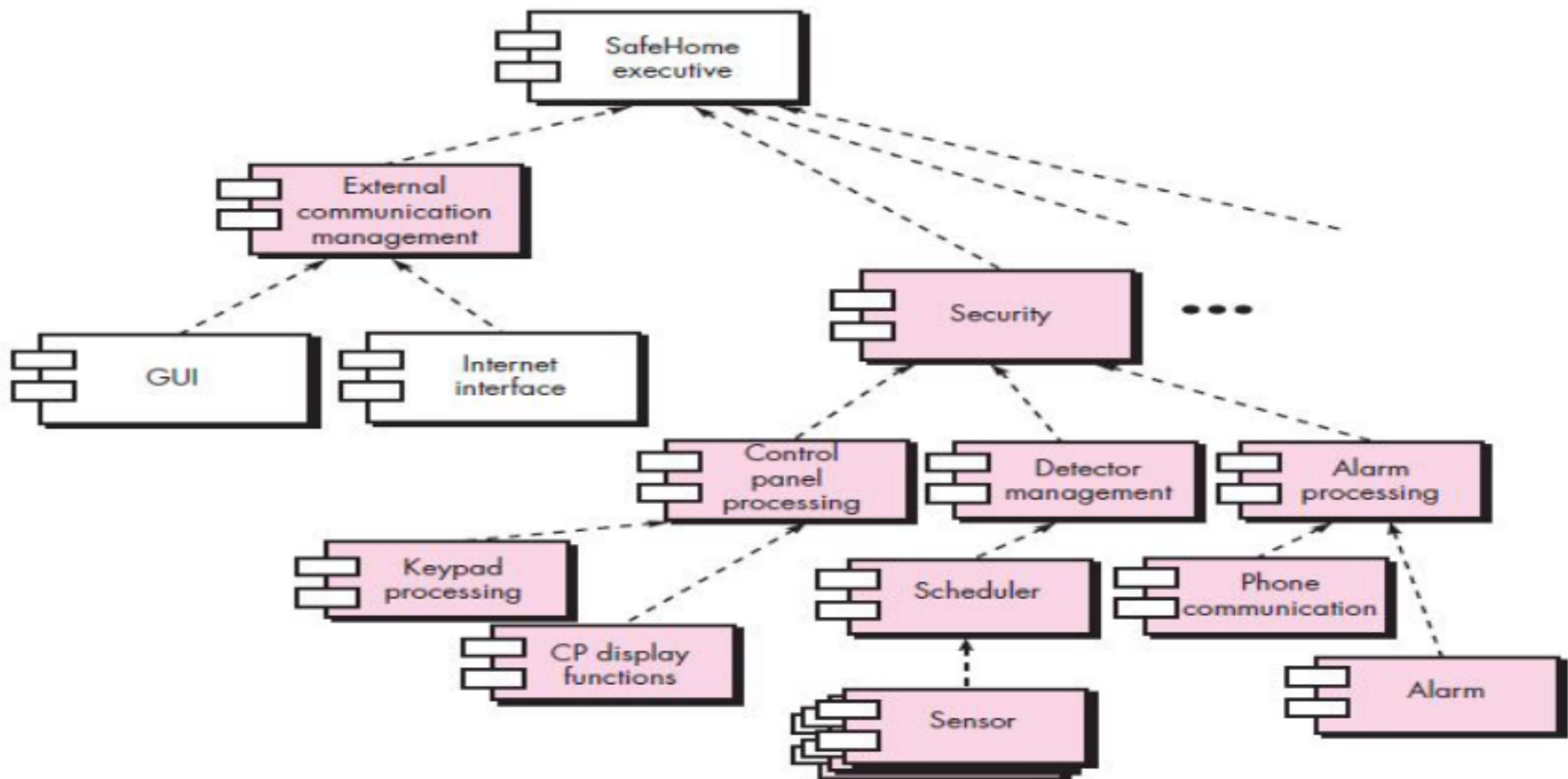
Archetypes – their methods



3. Refine the Architecture into Components

- Based on the archetypes, the architectural designer refines the software architecture into components to illustrate the overall structure and architectural style of the system
- These components are derived from various sources
 - The application domain provides application components, which are the domain classes in the analysis model that represent entities in the real world
 - The infrastructure domain provides design components (i.e., design classes) that enable application components but have no business connection
 - Examples: memory management, communication, database, and task management
 - The interfaces in the ACD imply one or more specialized components that process the data that flow across the interface
- A UML class diagram can represent the classes of the refined architecture and their relationships

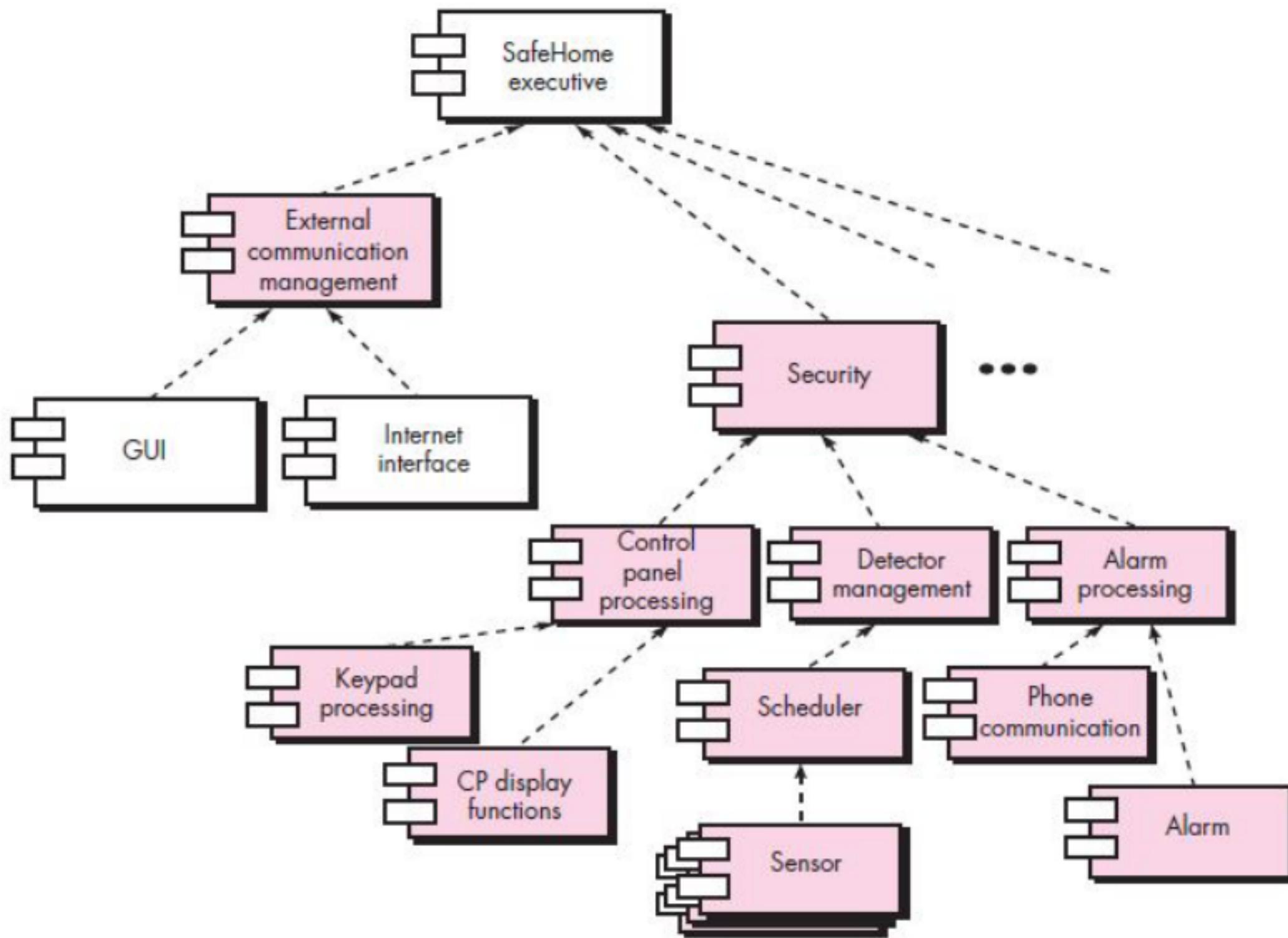
Overall architectural structure for *SafeHome* with top-level components



4. Describe Instantiations of the System

- An actual instantiation of the architecture is developed by applying it to a specific problem
- This demonstrates that the architectural structure, style and components are appropriate
- A UML component diagram can be used to represent this instantiation

An instantiation of the security function with component elaboration



Approach C: Assessing Architectural Complexity

- The overall complexity of a software architecture can be assessed by considering the dependencies between components within the architecture
- These dependencies are driven by the information and control flow within a system
- Three types of dependencies
 - Sharing dependency $U \bowtie \bowtie \bowtie \bowtie V$
 - Represents a dependency relationship among consumers who use the same source or producer
 - Flow dependency $\bowtie U \bowtie V \bowtie$
 - Represents a dependency relationship between producers and consumers of resources
 - Constrained dependency $U \text{ "XOR" } V$
 - Represents constraints on the relative flow of control among a set of activities such as mutual exclusion between two components

Component-Level Design

- Introduction
- The software component
- Designing class-based components
- Designing conventional components

Introduction

Background

- Component-level design occurs after the first iteration of the architectural design
- It strives to create a design model from the analysis and architectural models
 - The translation can open the door to subtle errors that are difficult to find and correct later
 - “Effective programmers should not waste their time debugging – they should not introduce bugs to start with.” **Edsgar Dijkstra**
- A component-level design can be represented using some intermediate representation (e.g. graphical, tabular, or text-based) that can be translated into source code
- The design of data structures, interfaces, and algorithms should conform to well-established guidelines to help us avoid the introduction of errors

The Software Component

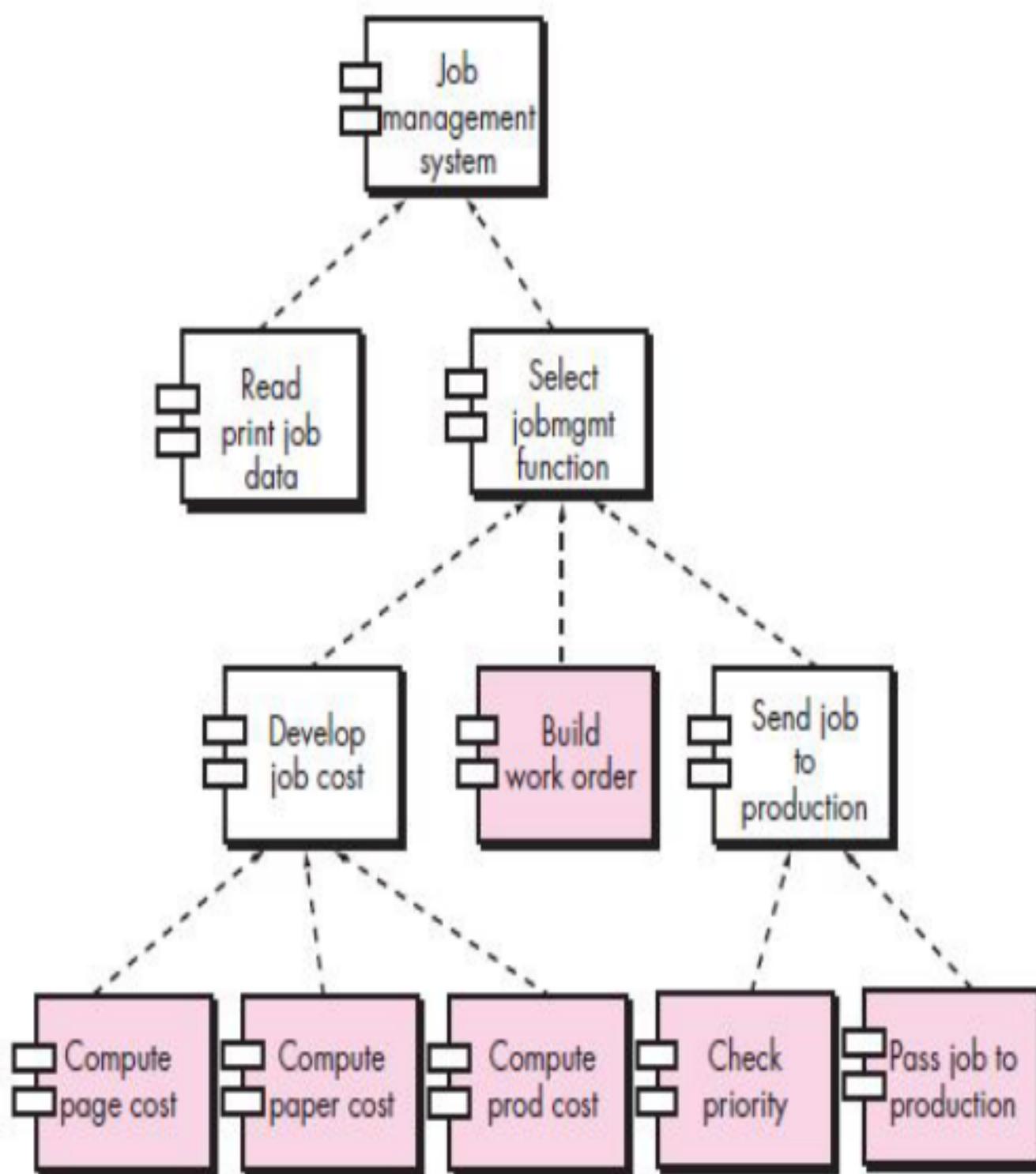
Defined

- A software component is a modular building block for computer software. Object Management Group (OMG) UML spec.
 - It is a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces
- A component communicates and collaborates with
 - Other components
 - Entities outside the boundaries of the system
- Three different views of a component
 - An object-oriented view
 - A conventional view
 - A process-related view

Object-oriented View

- A component is viewed as a set of one or more collaborating classes
- Each problem domain (i.e., analysis) class and infrastructure (i.e., design) class is elaborated to identify all attributes and operations that apply to its implementation
 - This also involves defining the interfaces that enable classes to communicate and collaborate
- This elaboration activity is applied to every component defined as part of the architectural design
- Once this is completed, the following steps are performed
 - Provide further elaboration of each attribute, operation, and interface
 - Specify the data structure appropriate for each attribute
 - Design the algorithmic detail required to implement the processing logic associated with each operation
 - Design the mechanisms required to implement the interface to include the messaging that occurs between objects

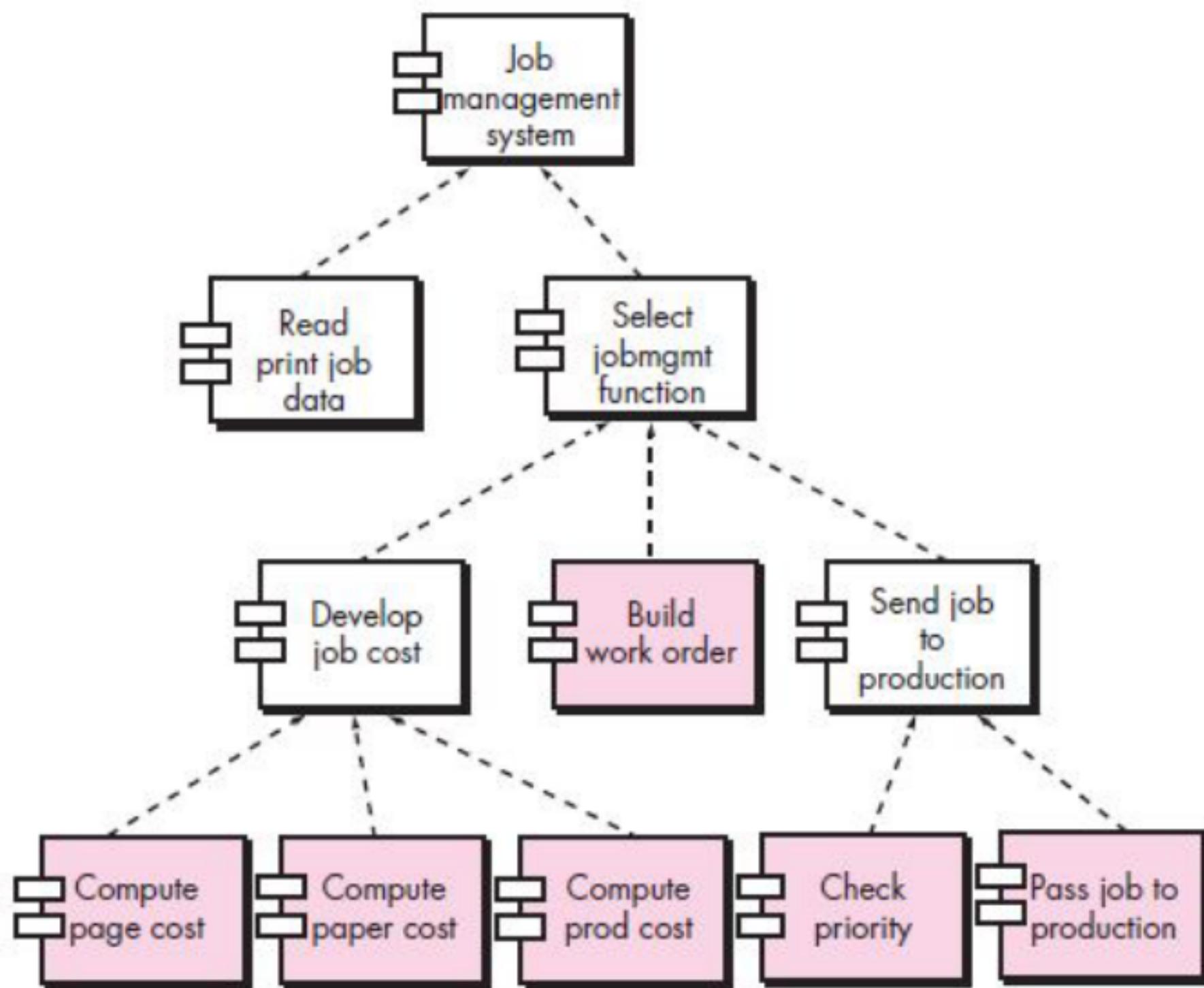
Elaboration of a design component



Conventional/Traditional View

- A component is viewed as a functional element (i.e., a module) of a program that incorporates
 - The processing logic
 - The internal data structures that are required to implement the processing logic
 - An interface that enables the component to be invoked and data to be passed to it
- A component serves one of the following roles
 - A control component that coordinates the invocation of all other problem domain components
 - A problem domain component that implements a complete or partial function that is required by the customer
 - An infrastructure component that is responsible for functions that support the processing required in the problem domain

Structure chart for a traditional system



Component-level design for *ComputePageCost*

Conditions	Rules					
	1	2	3	4	5	6
Regular customer	T	T				
Silver customer			T	T		
Gold customer					T	T
Special discount	F	T	F	T	F	T
Actions						
No discount	✓					
Apply 8 percent discount			✓	✓		
Apply 15 percent discount				✓	✓	
Apply additional x percent discount		✓		✓		✓

Conventional View (continued)

- Conventional software components are derived from the data flow diagrams (DFDs) in the analysis model
 - Each transform bubble (i.e., module) represented at the lowest levels of the DFD is mapped into a module hierarchy
 - Control components reside near the top
 - Problem domain components and infrastructure components migrate toward the bottom
 - Functional independence is strived for between the transforms
- Once this is completed, the following steps are performed for each transform
 - Define the interface for the transform (the order, number and types of the parameters)
 - Define the data structures used internally by the transform
 - Design the algorithm used by the transform (using a stepwise refinement approach)

Process-related View

- Emphasis is placed on building systems from existing components maintained in a library rather than creating each component from scratch
 - As the software architecture is formulated, components are selected from the library and used to populate the architecture
 - Because the components in the library have been created with reuse in mind, each contains the following:
 - A complete description of their interface
 - The functions they perform
 - The communication and collaboration they require
- Component-based Software Engineering

Designing Class-Based Components

Component-level Design Principles

- Open-closed principle (OCP)

- A module or component should be open for extension but closed for modification
- The designer should specify the component in a way that allows it to be extended without the need to make internal code or design modifications to the existing parts of the component

Conditions	Rules					
	1	2	3	4	5	6
Regular customer	T	T				
Silver customer			T	T		
Gold customer					T	T
Special discount	F	T	F	T	F	T
Actions						
	No discount	✓				
Apply 8 percent discount			✓	✓		
Apply 15 percent discount					✓	✓
Apply additional x percent discount		✓		✓		✓

Component-level Design Principles

- Liskov substitution principle (LSP)
 - Subclasses should be substitutable for their base classes
 - A component that uses a base class should continue to function properly if a subclass of the base class is passed to the component instead
- Dependency inversion principle (DIP)
 - Depend on abstractions (i.e., interfaces); do not depend on concretions
 - The more a component depends on other concrete components (rather than on the interfaces) the more difficult it will be to extend
- Interface segregation principle (ISP)
 - Many client-specific interfaces are better than one general purpose interface
 - For a server class, specialized interfaces should be created to serve major categories of clients
 - Only those operations that are relevant to a particular category of clients should be specified in the interface

Component Packaging Principles

- Release reuse equivalency principle (REP)
 - The granularity of reuse is the granularity of release
 - Group the reusable classes into packages that can be managed, upgraded, and controlled as newer versions are created
- Common closure principle (CCP)
 - Classes that change together belong together
 - Classes should be packaged cohesively; they should address the same functional or behavioral area on the assumption that if one class experiences a change then they all will experience a change
- Common reuse principle (CRP)
 - Classes that aren't reused together should not be grouped together
 - Classes that are grouped together may go through unnecessary integration and testing when they have

Component-Level Design Guidelines

- Components
 - Establish naming conventions for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model
 - Obtain architectural component names from the problem domain and ensure that they have meaning to all stakeholders who view the architectural model (e.g., Calculator)
 - Use infrastructure component names that reflect their implementation-specific meaning (e.g., Stack)
- Dependencies and inheritance in UML
 - Model any dependencies from left to right and inheritance from top (base class) to bottom (derived classes)
 - Consider modeling any component dependencies as interfaces rather than representing them as a direct component-to-component dependency

Cohesion

- Cohesion is the “single-mindedness’ of a component
- It implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself
- The objective is to keep cohesion as **high** as possible
- The kinds of cohesion can be ranked in order from highest (best) to lowest (worst)
 - Functional
 - A module performs one and only one computation and then returns a result
 - Layer
 - A higher layer component accesses the services of a lower layer component
 - Communicational
 - All operations that access the same data are defined within one class

Cohesion (continued)

- Kinds of cohesion (continued)
 - Sequential
 - Components or operations are grouped in a manner that allows the first to provide input to the next and so on in order to implement a sequence of operations
 - Procedural
 - Components or operations are grouped in a manner that allows one to be invoked immediately after the preceding one was invoked, even when no data passed between them
 - Temporal
 - Operations are grouped to perform a specific behavior or establish a certain state such as program start-up or when an error is detected
 - Utility
 - Components, classes, or operations are grouped within the same category because of similar general functions but are otherwise unrelated to each other

Coupling

- As the amount of communication and collaboration increases between operations and classes, the complexity of the computer-based system also increases
- As complexity rises, the difficulty of implementing, testing, and maintaining software also increases
- Coupling is a qualitative measure of the degree to which operations and classes are connected to one another
- The objective is to keep coupling as low as possible

(More on next slide)

Coupling (continued)

- The kinds of coupling can be ranked in order from lowest (best) to highest (worst)
 - Data coupling
 - Operation A() passes one or more atomic data operands to operation B(); the less the number of operands, the lower the level of coupling
 - Stamp coupling
 - A whole data structure or class instantiation is passed as a parameter to an operation
 - Control coupling
 - Operation A() invokes operation B() and passes a control flag to B that directs logical flow within B()
 - Consequently, a change in B() can require a change to be made to the meaning of the control flag passed by A(), otherwise an error may result
 - Common coupling
 - A number of components all make use of a global variable, which can lead to uncontrolled error propagation and unforeseen side effects
 - Content coupling
 - One component secretly modifies data that is stored internally in another component

(More on next slide)

Coupling (continued)

- Other kinds of coupling (unranked)
 - Subroutine call coupling
 - When one operation is invoked it invokes another operation within side of it
 - Type use coupling
 - Component A uses a data type defined in component B, such as for an instance variable or a local variable declaration
 - If/when the type definition changes, every component that declares a variable of that data type must also change
 - Inclusion or import coupling
 - Component A imports or includes the contents of component B
 - External coupling
 - A component communicates or collaborates with infrastructure components that are entities external to the software (e.g., operating system functions, database functions, networking functions)

Conducting Component-Level Design

- Identify all design classes that correspond to the problem domain as defined in the analysis model and architectural model
- Identify all design classes that correspond to the infrastructure domain
 - These classes are usually not present in the analysis or architectural models
 - These classes include GUI components, operating system components, data management components, networking components, etc.
- Elaborate all design classes that are not acquired as reusable components
 - Specify message details (i.e., structure) when classes or components collaborate
 - Identify appropriate interfaces (e.g., abstract classes) for each component
 - Elaborate attributes and define data types and data structures required to implement them (usually in the planned implementation language)
 - Describe processing flow within each operation in detail by means of pseudocode or UML activity diagrams

(More on next slide)

Conducting Component-Level Design (continued)

- Describe persistent data sources (databases and files) and identify the classes required to manage them
- Develop and elaborate behavioral representations for a class or component
 - This can be done by elaborating the UML state diagrams created for the analysis model and by examining all use cases that are relevant to the design class
- Elaborate deployment diagrams to provide additional implementation detail
 - Illustrate the location of key packages or classes of components in a system by using class instances and designating specific hardware and operating system environments
- Factor every component-level design representation and always consider alternatives
 - Experienced designers consider all (or most) of the alternative design solutions before settling on the final design model
 - The final decision can be made by using established design principles and guidelines

Designing Conventional/ Traditional Components

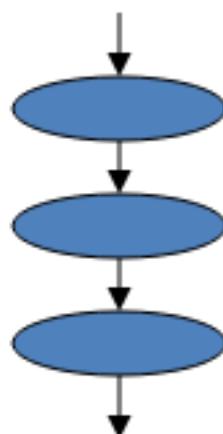
Introduction

- Proposed by Edsger Dijkstra in late 1960s.
- Conventional design constructs emphasize the maintainability of a functional/procedural program
 - Sequence, condition, and repetition
- Each construct has a predictable logical structure where control enters at the top and exits at the bottom, enabling a maintainer to easily follow the procedural flow
- Use of limited number of logical constructs also contributes to a human understanding process that psychologists call “**chunking**”

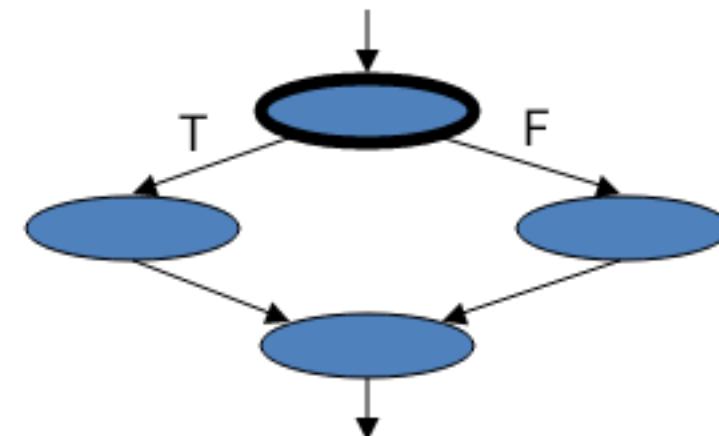
Introduction

- Various notations depict the use of these constructs
 - Graphical design notation
 - Sequence, if-then-else, selection, repetition (see next slide)
 - Tabular design notation (see upcoming slide)
 - Program Design Language (PDL)
 - Similar to a programming language; however, it uses narrative text embedded directly within the program statements

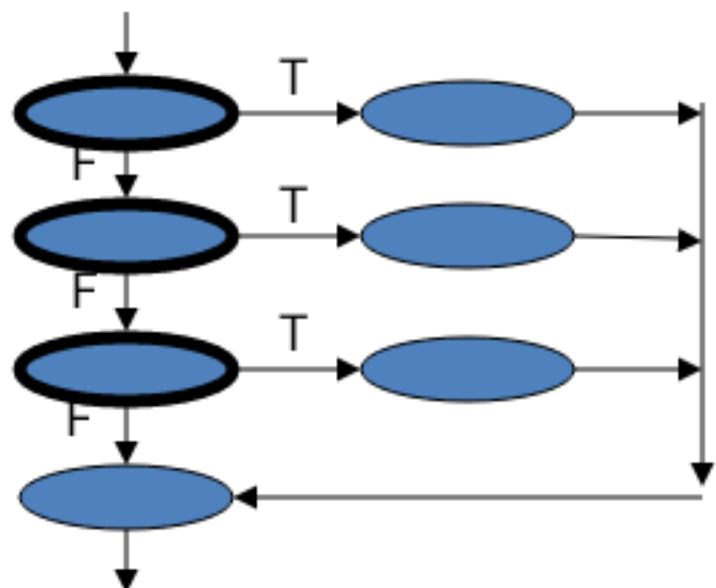
Graphical Design Notation



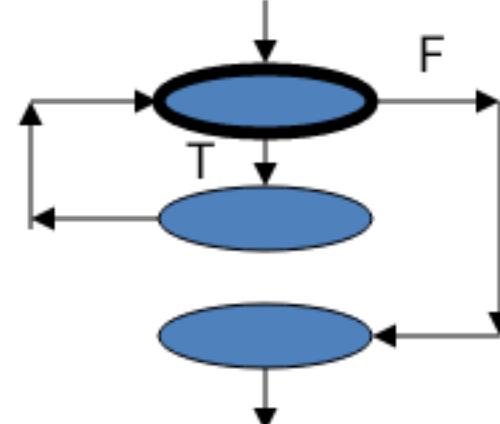
Sequence



If-then-else



Selection



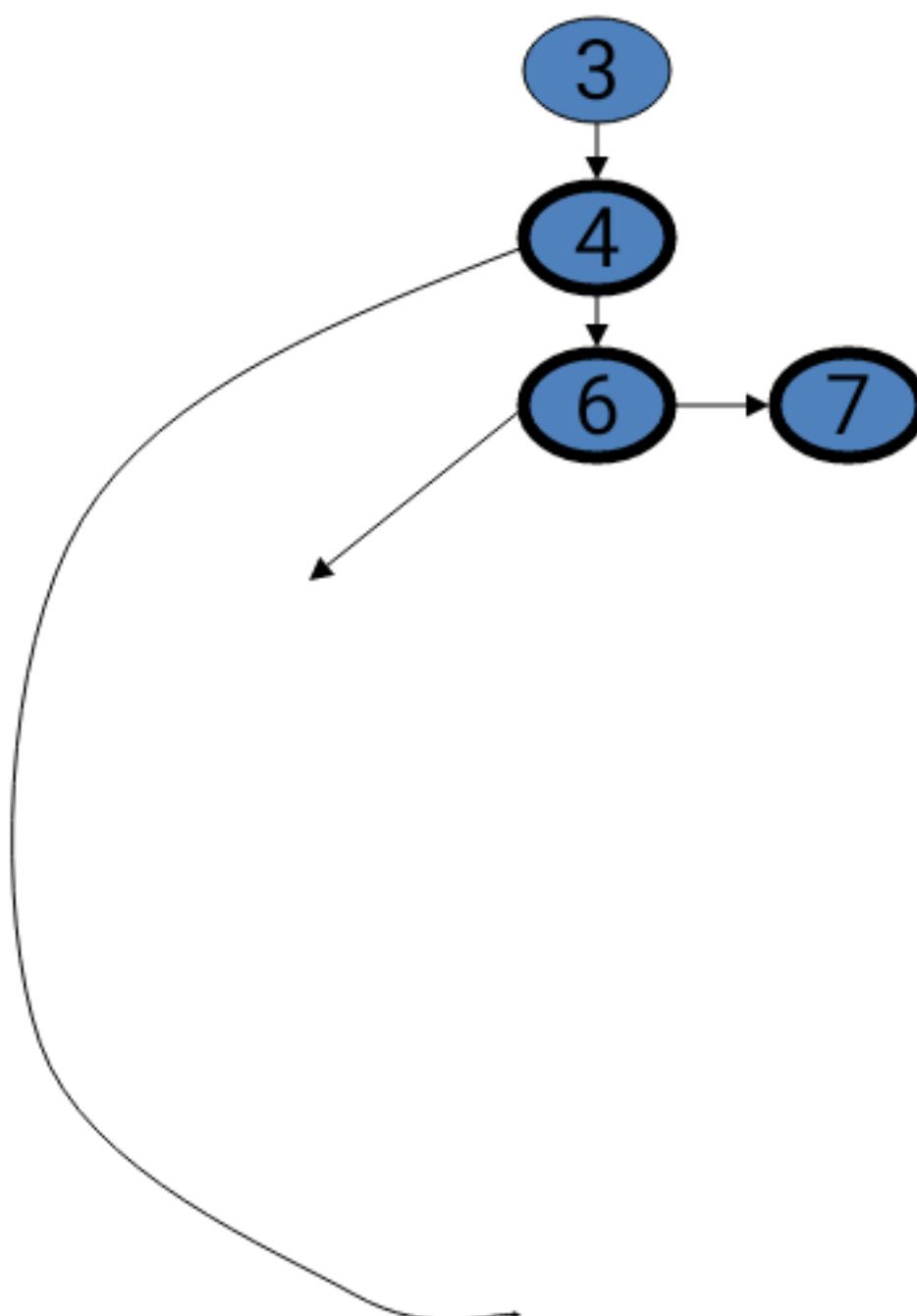
Repetition

Graphical Example used for Algorithm Analysis

```
1 int functionZ(int y)
2 {
3     int x = 0;
4
5     while (x <= (y * y))
6     {
7         if ((x % 11 == 0) &&
8             (x % y == 0))
9         {
10            printf("%d", x);
11            x++;
12        } // End if
13        else if ((x % 7 == 0) ||
14                  (x % y == 1))
15        {
16            printf("%d", y);
17            x = x + 2;
18        } // End else
19    } // End while
20
21    printf("\n");
22} // End functionZ
```

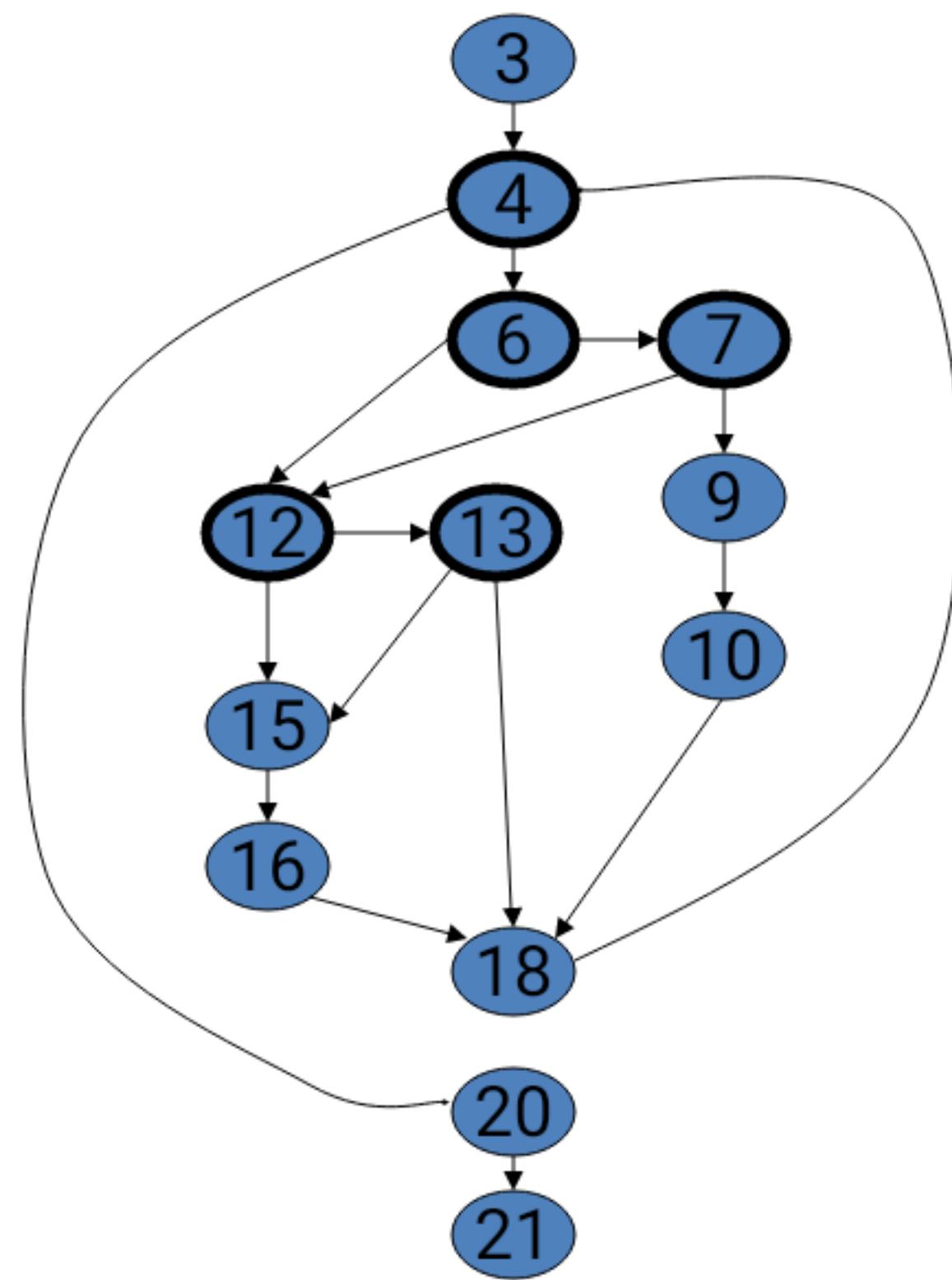
Graphical Example used for Algorithm Analysis

```
1 int functionZ(int y)
2 {
3     int x = 0;
4
5     while (x <= (y * y))
6     {
7         if ((x % 11 == 0) &&
8             (x % y == 0))
9         {
10            printf("%d", x);
11            x++;
12        } // End if
13        else if ((x % 7 == 0) ||
14                  (x % y == 1))
15        {
16            printf("%d", y);
17            x = x + 2;
18        } // End else
19    } // End while
20
21    printf("\n");
22} // End functionZ
```



Graphical Example used for Algorithm Analysis

```
1 int functionZ(int y)
2 {
3     int x = 0;
4
5     while (x <= (y * y))
6     {
7         if ((x % 11 == 0) &&
8             (x % y == 0))
9         {
10            printf("%d", x);
11            x++;
12        } // End if
13        else if ((x % 7 == 0) ||
14                  (x % y == 1))
15        {
16            printf("%d", y);
17            x = x + 2;
18        } // End else
19    } // End while
20
21    printf("\n");
22 } // End functionZ
```



Tabular Design Notation

- List all actions that can be associated with a specific procedure (or module)
- List all conditions (or decisions made) during execution of the procedure
- Associate specific sets of conditions with specific actions, eliminating impossible combinations of conditions; alternatively, develop every possible permutation of conditions
- Define rules by indicating what action(s) occurs for a set of conditions

Program Design Language

Program design language (PDL), also called *structured English* or *pseudocode*, incorporates the logical structure of a programming language with the free-form expressive ability of a natural language (e.g., English).

Refer text book for an example of *SafeHome* Security function.
(More on next slide)

Tabular Design Notation

Conditions	Rules					
	1	2	3	4	5	6
Regular customer	T	T				
Silver customer			T	T		
Gold customer					T	T
Special discount	F	T	F	T	F	T
Actions						
	No discount	✓				
Apply 8 percent discount		✓	✓			
Apply 15 percent discount				✓	✓	
Apply additional x percent discount	✓		✓		✓	

Quality Concepts

Slide Set to accompany
Software Engineering: A Practitioner's Approach, 7/e
by Roger S. Pressman

Software Quality

- In 2005, *ComputerWorld* [Hil05] lamented that
 - “bad software plagues nearly every organization that uses computers, causing lost work hours during computer downtime, lost or corrupted data, missed sales opportunities, high IT support and maintenance costs, and low customer satisfaction.
- A year later, *InfoWorld* [Fos06] wrote about the
 - “the sorry state of software quality” reporting that the quality problem had not gotten any better.
- Today, software quality remains an issue, but who is to blame?
 - Customers blame developers, arguing that sloppy practices lead to low-quality software.
 - Developers blame customers (and other stakeholders), arguing that irrational delivery dates and a continuing stream of changes force them to deliver software before it has been fully validated.

Quality

- The *American Heritage Dictionary* defines *quality* as
 - “a characteristic or attribute of something.”
- For software, two kinds of quality may be encountered:
 - **Quality of design** encompasses requirements, specifications, and the design of the system.
 - **Quality of conformance** is an issue focused primarily on implementation.
 - **User satisfaction** = compliant product + good quality + delivery within budget and schedule

Quality—A Philosophical View

- Robert Persig [Per74] commented on the thing we call *quality*.
 - Quality . . . you know what it is, yet you don't know what it is. But that's self-contradictory. But some things are better than others, that is, they have more quality. But when you try to say what the quality is, apart from the things that have it, it all goes poof! There's nothing to talk about. But if you can't say what Quality is, how do you know what it is, or how do you know that it even exists? If no one knows what it is, then for all practical purposes it doesn't exist at all. But for all practical purposes it really does exist. What else are the grades based on? Why else would people pay fortunes for some things and throw others in the trash pile? Obviously some things are better than others . . . but what's the betterness? . . . So round and round you go, spinning mental wheels and nowhere finding anyplace to get traction. What the hell is Quality? **What is it?**

Quality—A Pragmatic View

- The *transcendental view* argues (like Persig) that quality is something that you immediately recognize, but cannot explicitly define.
- The *user view* sees quality in terms of an end-user's specific goals. If a product meets those goals, it exhibits quality.
- The *manufacturer's view* defines quality in terms of the original specification of the product. If the product conforms to the spec, it exhibits quality.
- The *product view* suggests that quality can be tied to inherent characteristics (e.g., functions and features) of a product.
- Finally, the *value-based view* measures quality based on how much a customer is willing to pay for a product. In reality, quality encompasses all of these views and more.

Software Quality

- Software quality can be defined as:
 - *An effective software process applied in a manner that creates a useful product that provides measurable value for those who produce it and those who use it.*
- This definition has been adapted from [Bes04] and replaces a more manufacturing-oriented view presented in earlier editions of this book.

Effective Software Process

- An *effective software process* establishes the infrastructure that supports any effort at building a high quality software product.
- The management aspects of process create the checks and balances that help avoid project chaos—a key contributor to poor quality.
- Software engineering practices allow the developer to analyze the problem and design a solid solution—both critical to building high quality software.
- Finally, umbrella activities such as change management and technical reviews have as much to do with quality as any other part of software engineering practice.

Useful Product

- A *useful product* delivers the content, functions, and features that the end-user desires
- But as important, it delivers these assets in a reliable, error free way.
- A useful product always satisfies those requirements that have been explicitly stated by stakeholders.
- In addition, it satisfies a set of implicit requirements (e.g., ease of use) that are expected of all high quality software.

Adding Value

- By *adding value for both the producer and user* of a software product, high quality software provides benefits for the software organization and the end-user community.
- The software organization gains added value because high quality software requires less maintenance effort, fewer bug fixes, and reduced customer support.
- The user community gains added value because the application provides a useful capability in a way that expedites some business process.
- The end result is:
 - (1) greater software product revenue,
 - (2) better profitability when an application supports a business process, and/or
 - (3) improved availability of information that is crucial for the business.

Quality Dimensions

- David Garvin [Gar87]:
 - **Performance Quality.** Does the software deliver all content, functions, and features that are specified as part of the requirements model in a way that provides value to the end-user?
 - **Feature quality.** Does the software provide features that surprise and delight first-time end-users?
 - **Reliability.** Does the software deliver all features and capability without failure? Is it available when it is needed? Does it deliver functionality that is error free?
 - **Conformance.** Does the software conform to local and external software standards that are relevant to the application? Does it conform to de facto design and coding conventions? For example, does the user interface conform to accepted design rules for menu selection or data input?

Quality Dimensions

- **Durability.** Can the software be maintained (changed) or corrected (debugged) without the inadvertent generation of unintended side effects? Will changes cause the error rate or reliability to degrade with time?
- **Serviceability.** Can the software be maintained (changed) or corrected (debugged) in an acceptably short time period. Can support staff acquire all information they need to make changes or correct defects?
- **Aesthetics.** Most of us would agree that an aesthetic entity has a certain elegance, a unique flow, and an obvious “presence” that are hard to quantify but evident nonetheless.
- **Perception.** In some situations, you have a set of prejudices that will influence your perception of quality.

Other Views

- **McCall's Quality Factors** (*SEPA*, Section 14.2.2)
- **ISO 9126 Quality Factors** (*SEPA*, Section 14.2.3)
- **Targeted Factors** (*SEPA*, Section 14.2.4)

The Software Quality Dilemma

- If you produce a software system that has terrible quality, you lose because no one will want to buy it.
- If on the other hand you spend infinite time, extremely large effort, and huge sums of money to build the absolutely perfect piece of software, then it's going to take so long to complete and it will be so expensive to produce that you'll be out of business anyway.
- Either you missed the market window, or you simply exhausted all your resources.
- So people in industry try to get to that magical middle ground where the product is good enough not to be rejected right away, such as during evaluation, but also not the object of so much perfectionism and so much work that it would take too long or cost too much to complete.
[Ven03]

“Good Enough” Software

- Good enough software delivers high quality functions and features that end-users desire, but at the same time it delivers other more obscure or specialized functions and features that contain known bugs.
- Arguments *against* “good enough.”
 - It is true that “good enough” may work in some application domains and for a few major software companies. After all, if a company has a large marketing budget and can convince enough people to buy version 1.0, it has succeeded in locking them in.
 - If you work for a small company be wary of this philosophy. If you deliver a “good enough” (buggy) product, you risk permanent damage to your company’s reputation.
 - You may never get a chance to deliver version 2.0 because bad buzz may cause your sales to plummet and your company to fold.
 - If you work in certain application domains (e.g., real time embedded software, application software that is integrated with hardware) can be negligent and open your company to expensive litigation.

Cost of Quality

- *Prevention costs* include
 - ?
- *Internal failure costs* include
 - ?
- *External failure costs* are
 - ?

Cost of Quality

- *Prevention costs* include

- quality planning
- formal technical reviews
- test equipment
- Training

- *Internal failure costs* include

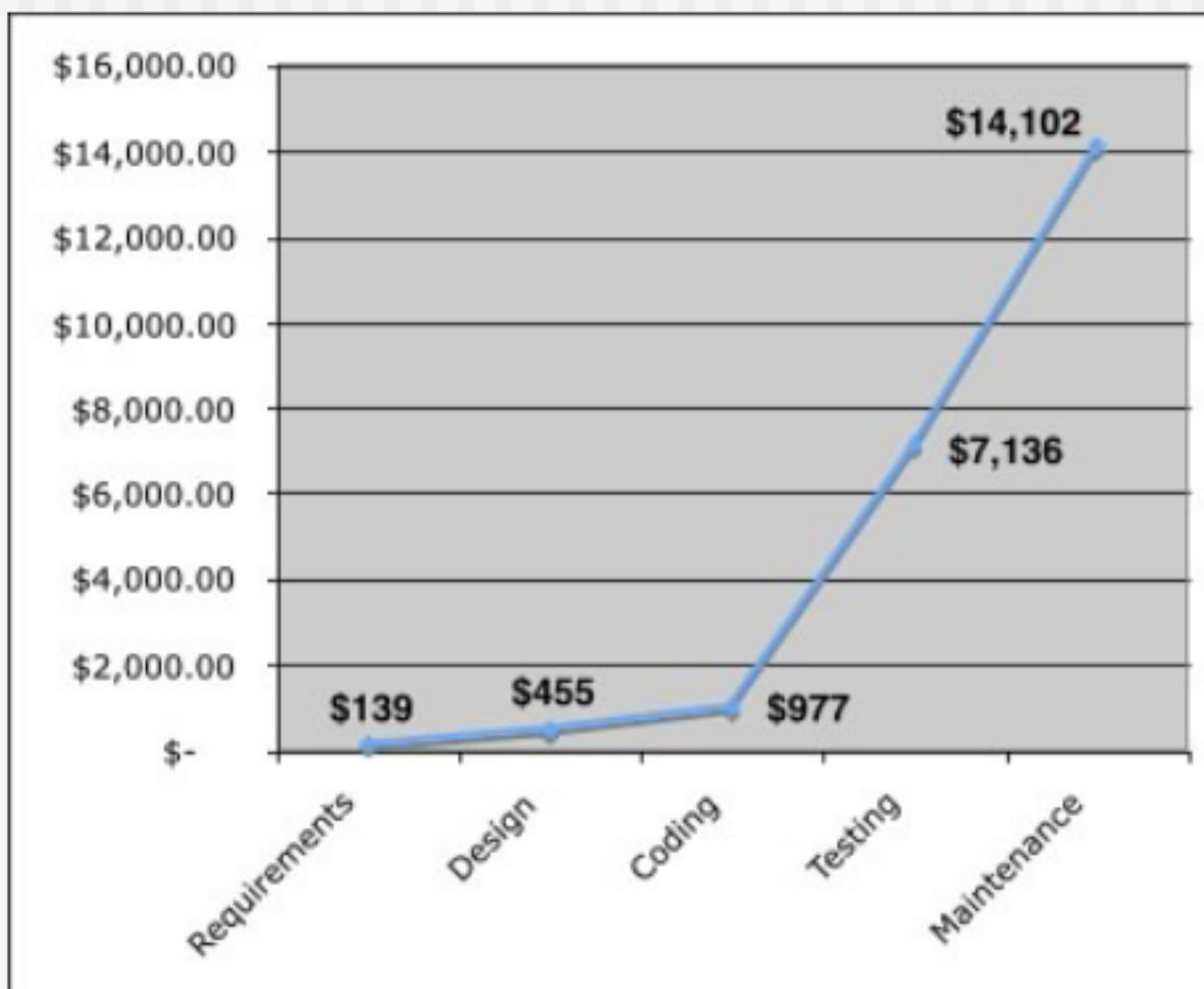
- rework
- repair
- failure mode analysis

- *External failure costs* are

- complaint resolution
- product return and replacement
- help line support
- warranty work

Cost

- The relative costs to find and repair an error or defect increase dramatically as we go from prevention to detection to internal failure to external failure costs.



Quality and Risk

- *"People bet their jobs, their comforts, their safety, their entertainment, their decisions, and their very lives on computer software. It better be right."* SEPA, Chapter 1
- Example:
 - *Throughout the month of November, 2000 at a hospital in Panama, 28 patients received massive overdoses of gamma rays during treatment for a variety of cancers. In the months that followed, five of these patients died from radiation poisoning and 15 others developed serious complications. What caused this tragedy? A software package, developed by a U.S. company, was modified by hospital technicians to compute modified doses of radiation for each patient.*

Negligence and Liability

- The story is all too common. A governmental or corporate entity hires a major software developer or consulting company to analyze requirements and then design and construct a software-based “system” to support some major activity.
 - The system might support a major corporate function (e.g., pension management) or some governmental function (e.g., healthcare administration or homeland security).
- Work begins with the best of intentions on both sides, but by the time the system is delivered, things have gone bad.
- The system is late, fails to deliver desired features and functions, is error-prone, and does not meet with customer approval.
- Litigation ensues.

Quality and Security

- Gary McGraw comments [Wil05]:
- “Software security relates entirely and completely to quality. You must think about **security, reliability, availability, dependability**—at the beginning, in the design, architecture, test, and coding phases, all through the software life cycle **[process]**. Even people aware of the software security problem have focused on late life-cycle stuff. The earlier you find the software problem, the better. And there are two kinds of software problems. One is bugs, which are implementation problems. The other is software flaws—architectural problems in the design. **People pay too much attention to bugs and not enough on flaws.**”

Achieving Software Quality

- Critical success factors:
 - Software Engineering Methods
 - Project Management Techniques
 - Quality Control
 - Quality Assurance

Implementation Issues

(Source: Richard Fairley: Software Engineering Concepts

Objective

- Translate Design specifications to Source code
- Write source code and internal documentation – conformance to its specification can be easily **verified** – debugging, testing, and modifications
- Hallmarks of good programs: **Clarity** and **Elegance**
- Obscurity, Cleverness and Complexity – Inadequate design and misdirected thinking.

Weinberg's experiment

Major objective	Outcome		Clarity of		Minimum number of statements	Minimum Dvmt. time
	Minimum memory	Output	Program			
Minimize primary memory	1	4	4		2	5
Maximize output readability	5	1	1-2		5	2-3
Maximize source text readability	3	2	1-2		3	4
Minimize number of statements	2	5	3		1	2-3
Minimize development time	4	3	5		4	1

1: Best

5: Worst

Structured Coding Techniques

- Objective: Linearize control flow through a computer program
- Enhance: **Readability of code** – To ease understanding debugging, testing, documentation, and modification of programs.
- Facilitates **formal verification of programs** - the process of using automatic proof procedures to establish that a computer program will do what it's supposed to.

1. Single Entry, Single Exit Constructs

- By Bohm and Jacopini in 1966
- To describe the control flow of every algorithm – Sequencing, Selection and Iteration
- Every Turning Machine program can be written in the same manner

Bohm-Jacopini Theorem:

Any single entry, single exit program segment that has all statement on some path from the entry to the exit can be specified using only sequencing, selection, and iteration.

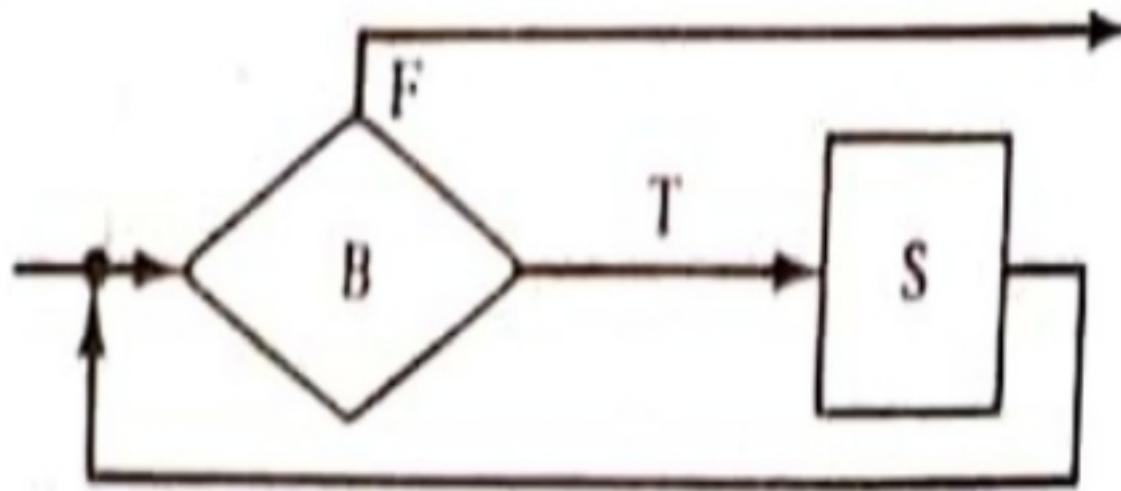
Linearity of control flow is retained, even with arbitrarily deep nesting of constructs.

Sequencing: S1; S2; S3;

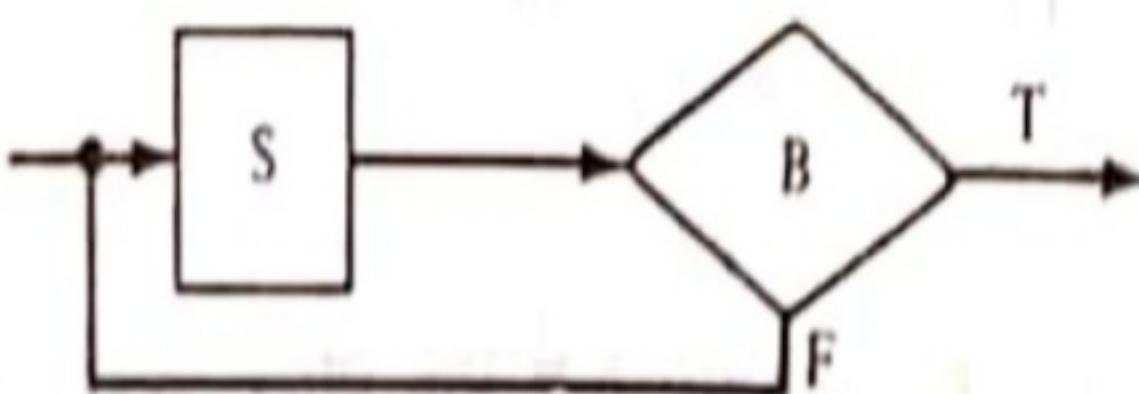
Selection: if B then S1 else S2

Sufficient set of constructs for structured coding

while B do S



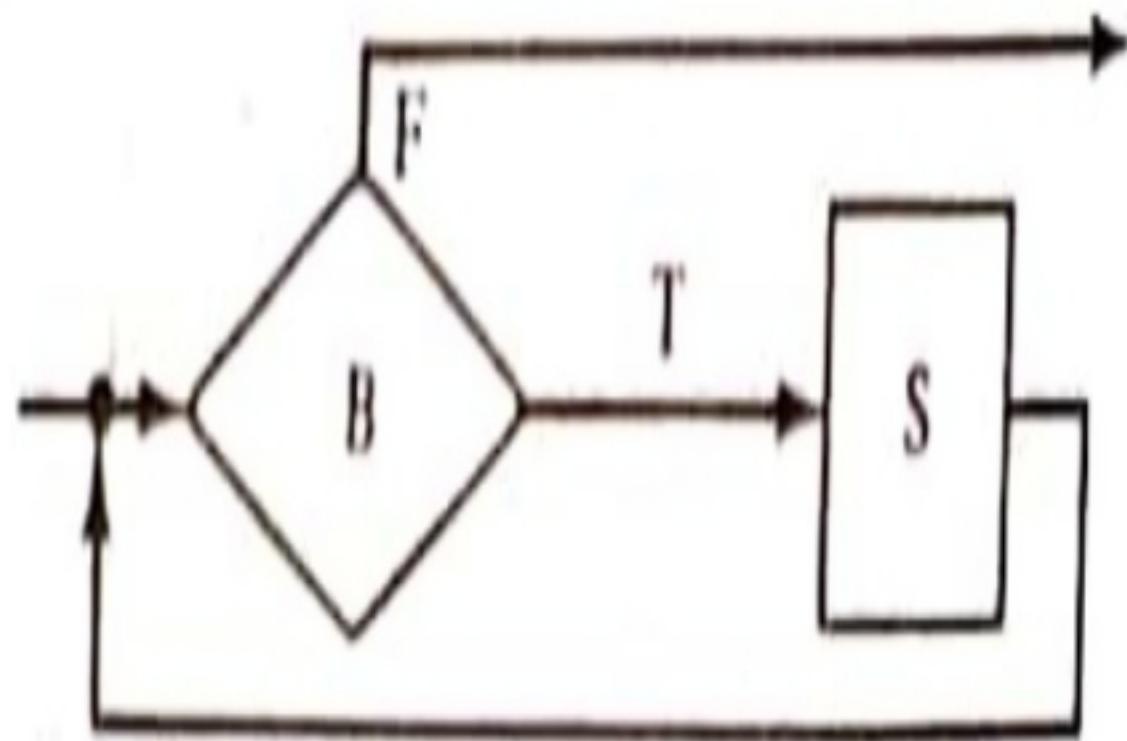
repeat S until B



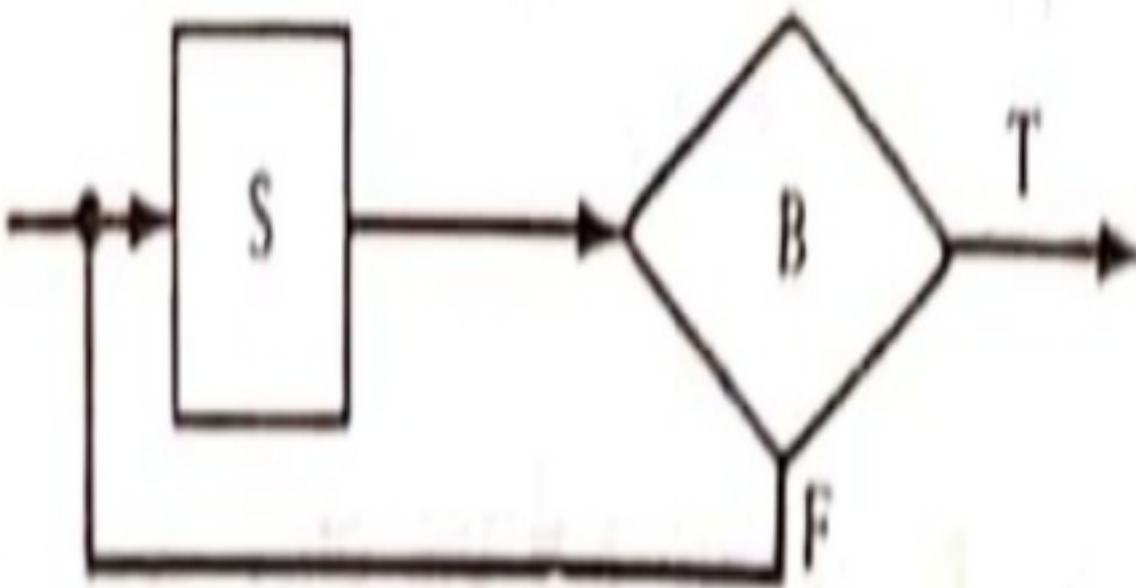
Increased notational convenience, increased readability, and increases efficiency

Pascal's control structures

while B do S

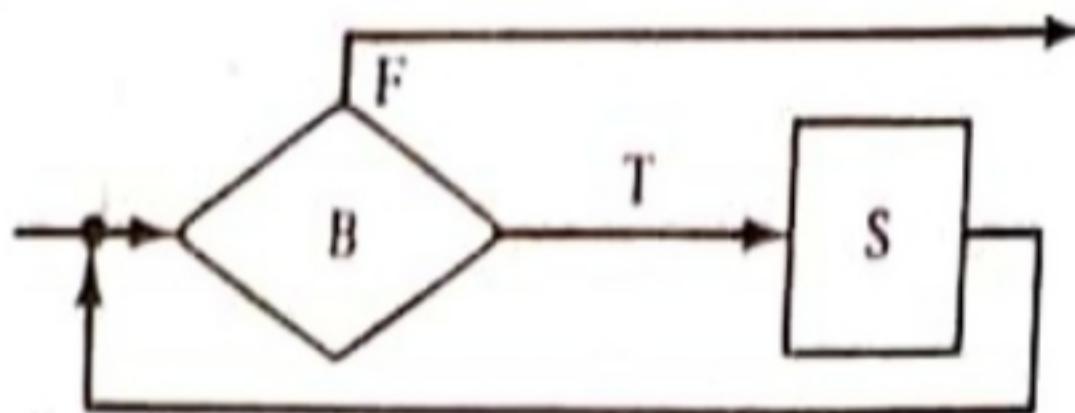


repeat S until B

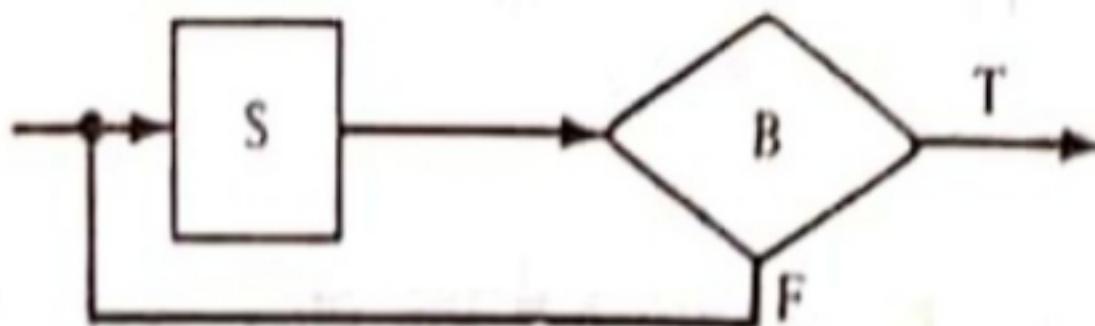


Pascal's control structures contd'

while B do S



repeat S until B



Pascal's control structures contd'

```
DO 10 I = 1,N  
:::::  
IF (COND1) GO TO 20  
:::::  
IF (COND2) GO TO 30  
:::::  
10 CONTINUE  
20 -- CODE FOR COND1 EXIT  
     GO TO 40  
30 -- CODE FOR COND2 EXIT  
40 CONTINUE
```

```
for I in 1..MAX loop  
:::::  
exit when COND1;  
:::::  
exit when COND2;  
:::::  
end loop;  
if (COND1) then  
  --code for COND1 exit  
elsif (COND2) then  
  --code for COND1 exit  
end if;  
  --code for normal exit
```

Pascal's control structures contd'

Repeat S1 until B:

S1;

while (not B) do S1;

A 'for' loop can be expressed as a sequence of the form:

S0;

while B loop Increased notational convenience, increased
 readability, and increases efficiency

 S1;

 S2;

end loop;

S0 – initializes the loop variable, B tests the limit, and S2 increments (or decrements) the loop variable;

S1 – body

Violation of Single Entry, Single Exit

Goal of Structured Coding: Clarity and Readability of source programs

Violation: Multiple loop exits, Error handling and Data encapsulation

Solutions: Boolean flag variable for each condition and test the flags on each iteration of the loop

Difficulties: Need for additional variable,
the additional tests on loop exit to determine
which condition caused the exit

Multiple loop exits in FORTRAN IV and Ada

```
DO 10 I = 1,N
::::
IF (COND1) GO TO 20
::::
IF (COND2) GO TO 30
::::
10 CONTINUE
20 -- CODE FOR COND1 EXIT
     GO TO 40
30 -- CODE FOR COND2 EXIT
40 CONTINUE
```

```
for I in 1..MAX loop
::::
exit when COND1;
::::
exit when COND2;
::::
end loop;
if (COND1) then
  —code for COND1 exit
elsif (COND2) then
  —code for COND2 exit
end if;
  —code for normal exit
```

Confusing Code??



3. Goto Statement

- Provides uncondition transfer of control and allows violation of the single entry, single exit.

```
L2: -- Code for Condition 2  
      goto L5  
L3: -- Code for Condition 3  
      goto L5  
L4: -- Code for all other conditions  
L5: Continue
```

In primitive programming languages, this goto version may be the best structure that can be achieved.

A preferable structure is the more compact form:

```
if (Condition 1) then  
    -- Code for Condition 1;  
elseif (Condition 2) then  
    -- Code for Condition 2;  
elseif (Condition 3) then  
    -- Code for Condition 3;  
else  
    -- Code for Condition 4;  
end if;
```

A complicated goto

```
L2: -- Code for Condition 2  
      goto L5  
L3: -- Code for Condition 3  
      goto L5  
L4: -- Code for all other conditions  
L5: Continue
```

In primitive programming languages, this goto version may be the best structure that can be achieved.

A preferable structure is the more compact form:

```
if (Condition 1) then  
    -- Code for Condition 1;  
elseif (Condition 2) then  
    -- Code for Condition 2;  
elseif (Condition 3) then  
    -- Code for Condition 3;  
else  
    -- Code for Condition 4;  
end if;
```

Dos of good coding style

type COLOR = (RED, GREEN, YELLOW, BLUE);

permits array indices and loop variables of the form

```
for I in GREEN..BLUE loop
    X(I) := 0;
end loop;
X(RED) := 0;
```

Subtypes of the form

type SMALL_INTEGER is INTEGER (1..10);

place range constraints on objects of type SMALL_INTEGER, and derived types of the form

type TEMP_IN_DEG_KELVIN is FLOAT;

type VELOCITY_IN_FPS is FLOAT;

Coding Style

- Manifested in the patterns used by a programmer to express a desired action or outcome.
- Goal: Easily understood, straightforward, elegant code.
- No single set of rules; but general guidelines are applicable.

Use of few standard control constructs:

Conditional and Unconditional branching in stylistic patterns to achieve the effects of if_then_else, while_do, repeat_until, case, etc.

Use of gotos:

Can be used in conjunction with the if statement to achieve the format and effect of structured coding constructs.

goto in Ada

```
<<Sort>>
for I in 1 .. N-1 loop
    if A(I) > A(I+1) then
        Exchange(A(I), A(I+1));
        goto Sort;
    end if;
end loop;
```

User-defined data types to model entities in the problem domain

`type COLOR = (RED, GREEN, YELLOW, BLUE);`
permits array indices and loop variables of the form

```
for I in GREEN..BLUE loop  
    X(I) := 0;  
end loop;  
X(RED) := 0;
```

Subtypes of the form

`type SMALL_INTEGER is INTEGER (1..10);`

place range constraints on objects of type `SMALL_INTEGER`, and derived types of the form

```
type TEMP_IN_DEG_KELVIN is FLOAT;  
type VELOCITY_IN_FPS is FLOAT;
```

Use of indentations

```
DO 10 I = 1,N  
DO 10 J = 1,N  
10 A(I,J) = (I/J)*(J/I)
```

Don'ts of good coding style

```
DO 10 I = 1,N  
DO 10 J = 1,N  
10 A(I,J) = (I/J)*(J/I)
```

Don't be too clever

This?

```
DO 10 I = 1,N
```

```
DO 10 J = 1,N
```

```
10 A(I,J) = (I/J)*(J/I)
```

or this?

```
DO 10 I = 1,N
```

```
DO 10 J = 1,N
```

```
10 A(I,J) = (I/J)*(J/I)
```

Avoid null then statements

- A null then statement is of the form

if B then;

else S;

which is equivalent to

if (not B) then S;

UNIT NAME: _____ PROGRAMMER: _____

ROUTINES INCLUDED: _____

SECTION	CONTENTS	DUE DATE	COMPLETED DATE	REVIEWER/DATE
1	RQMTS.			
2	ARCH. DESIGN			
3	DETAIL DESIGN			
4	TEST PLAN			
5	SOURCE CODE			
6	TEST RESULTS			
7	CHANGE REQUESTS			
8	NOTES			

RELEASE APPROVAL: _____ DATE: _____

Use of else_if instead of then_if

UNIT NAME: _____ PROGRAMMER: _____
ROUTINES INCLUDED: _____

SECTION	CONTENTS	DUE DATE	COMPLETED DATE	REVIEWER/DATE
1	RQMTS.			
2	ARCH. DESIGN			
3	DETAIL DESIGN			
4	TEST PLAN			
5	SOURCE CODE			
6	TEST RESULTS			
7	CHANGE REQUESTS			
8	NOTES			

RELEASE APPROVAL: _____ DATE: _____

Don't' nest too deeply

```
while B1 loop  
  if B2 ten  
    repeat S1  
      while B3 loop  
        if B4 then S2
```

Avoid obscure side effects

Side effect of a subprogram invocation is any change to the computational state that occurs as a result of calling the invoked subroutine.

E.g. `SORT(X,Y,Z);` - Transmission modes of X, Y,Z and the global variables modified by SORT can only be determined by examining the SORT routing.

Don't suboptimize - Not to waste time and effort worrying about situations that are handled in nonintuitive ways by the system.

Carefully examine routines having more than five formal parameters –

Fewer parameters (3 or 4) and fewer global variables improve the clarity and simplicity of subprograms.

Don't use an identifier for multiple purposes –

Results in ??

Good source: Ada, loop variables are inside the scope of the loop body.

Standards and Guidelines

- Coding standards are specifications for a preferred coding style.
- Artists, Poets, Musicians, dancers... Standards?
- For uniform quality

“Programming Guidelines” instead of Programming standards

Standards example:

Goto statements will not be used

The nesting depth of program constructs will not exceed five levels.

Rephrased Guidelines:

The use of goto statements should be avoided in normal circumstances.

The nesting depth of program constructs should be five or ₂₇ less in normal circumstances

Documentation Guidelines

Supporting Documents:

Requirements specification

Design documents

Test plans

User's manuals

Installation instructions and

Maintenance reports

Prepare documents in the same phase

Customer needs and constraints-> SRS-> Architectural design-> detailed design-> source code

Quality, quantity, timeliness, and utility of the supporting documents – measures for health and well being of a s/w project.

Program unit textbooks

- Unit of source code developed and or maintained by one person; that person is responsible for the unit.

Program Unit notebook – “**Unit development folder**”: consists of a cover sheet and several sections

Cover sheet – table of contents and sign-off sheet for the various milestones associated with the program unit.

Cover sheet for a program unit notebook

UNIT NAME: _____ PROGRAMMER: _____

ROUTINES INCLUDED: _____

SECTION	CONTENTS	DUe DATE	COMPLETED DATE	REVIEWER/DATE
1	RQMTS.			
2	ARCH. DESIGN			
3	DETAIL DESIGN			
4	TEST PLAN			
5	SOURCE CODE			
6	TEST RESULTS			
7	CHANGE REQUESTS			
8	NOTES			

RELEASE APPROVAL: _____ DATE: _____

Internal documentation

Typical format of subprogram and compilation unit prologues

1. Minimize the need for embedded comments by using:
 - Standard prologues
 - Structured programming constructs
 - Good coding style
 - Descriptive names from the problem domain for user-defined data types, variables, formal parameters, enumeration literals, subprograms, files, etc.
 - Self-documenting features of the implementation language, such as user-defined exceptions, user-defined data types, data encapsulation, etc.
2. Attach comments to blocks of code that:
 - Perform major data manipulations
 - Simulate structured control constructs using goto statements
 - Perform exception handling
3. Use problem domain terminology in the comments.
4. Use blank lines, borders, and indentation to highlight comments.
5. Place comments to the far right to document changes and revisions.
6. Don't use long, involved comments to document obscure, complex code. Rewrite the code.
7. Always be sure that comments and code agree with one another, and with the requirements and design specifications.

Commenting Conventions

1. Minimize the need for embedded comments by using:
 - Standard prologues
 - Structured programming constructs
 - Good coding style
 - Descriptive names from the problem domain for user-defined data types, variables, formal parameters, enumeration literals, subprograms, files, etc.
 - Self-documenting features of the implementation language, such as user-defined exceptions, user-defined data types, data encapsulation, etc.
2. Attach comments to blocks of code that:
 - Perform major data manipulations
 - Simulate structured control constructs using goto statements
 - Perform exception handling
3. Use problem domain terminology in the comments.
4. Use blank lines, borders, and indentation to highlight comments.
5. Place comments to the far right to document changes and revisions.
6. Don't use long, involved comments to document obscure, complex code. Rewrite the code.
7. Always be sure that comments and code agree with one another, and with the requirements and design specifications.

SOFTWARE QUALITY ASSURANCE

Slides adapted from various web sources with grateful acknowledgement of the many others who made their course materials freely available online.

SOFTWARE QUALITY ASSURANCE OUTLINE

Quality Assurance

- An Umbrella Activity – Software Metrics
- Software Standards

Product Quality

- Design Quality Metrics – Formal Approaches
- Program Quality Metrics

Project Quality

- Reviews
- Software Configuration Management

Process Quality

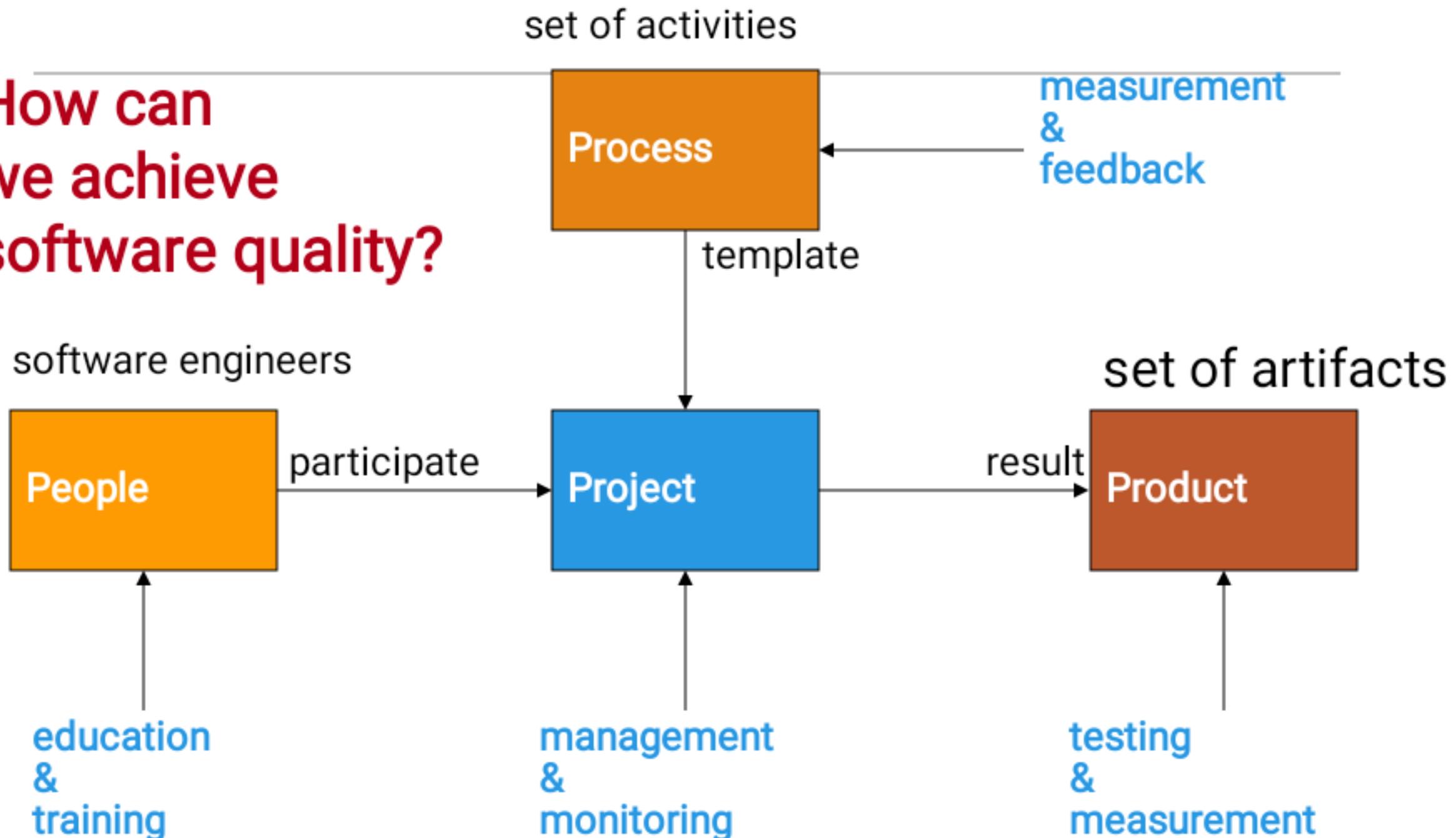
- ISO 9001/9000-3
- The SEI Capability Maturity Model

People Quality

- The People Capability Maturity Model

THE FOUR P's IN SOFTWARE DEVELOPMENT

How can
we achieve
software quality?

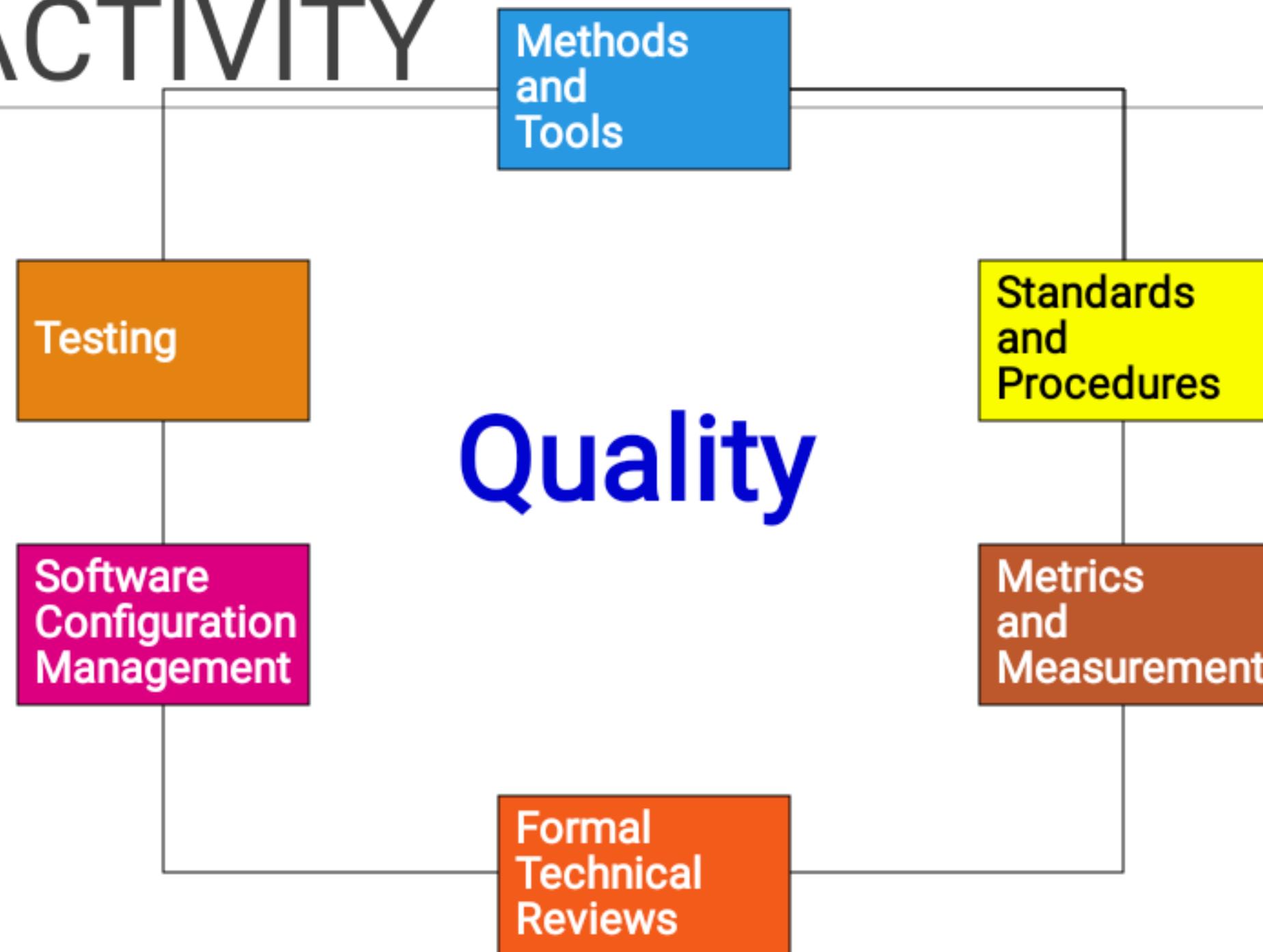


SOFTWARE QUALITY ASSURANCE (SQA)

Quality assurance consists of those procedures, techniques, and tools applied by professionals to ensure that a product meets or exceeds pre-specified standards during its development cycle. E.H. Bersoff

- It is an **essential activity** for any business that produces products used by others.
- It needs to be **planned** and **systematic**.
(It does not just happen)
- It needs to be **built into** the development process.
(A natural outcome of software engineering)
- **Continuous improvement** is the overall goal.

SQA – AN UMBRELLA ACTIVITY



SQA – AN UMBRELLA ACTIVITY

Includes all techniques used to improve the quality of the software:

1. **methods and tools** for System Requirements Capture, System Analysis, System Design, Implementation and Testing

- *to help ensure that we achieve a high quality system*

2. **standards and procedures** compliance

- *to help ensure repeatability of the development process*

3. **metrics and measurement** procedures

- *to help us improve the software development process*

4. **formal technical reviews** at each step

- *to help uncover quality problems and to sign-off*

5. **software configuration management** and **change control**

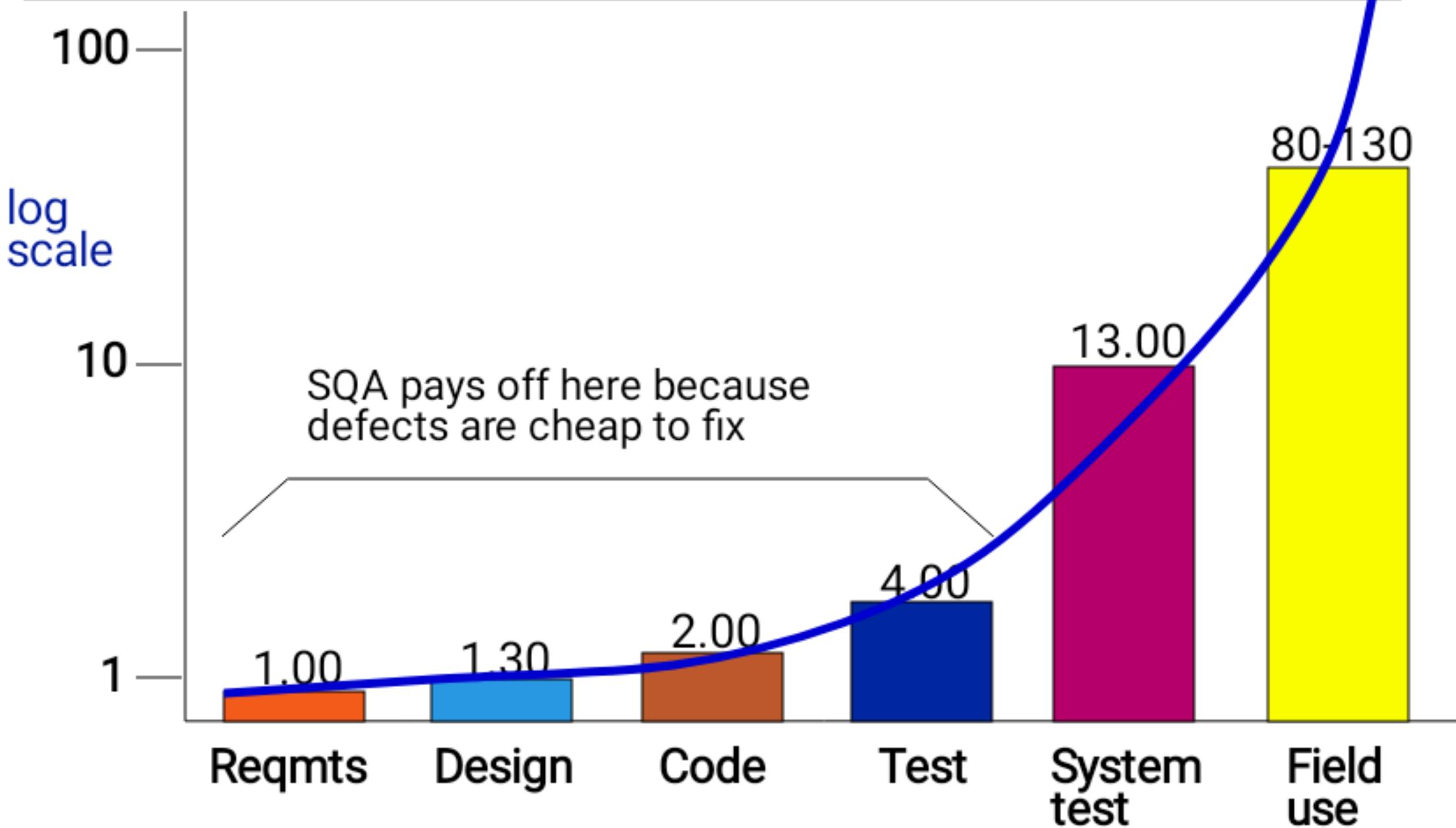
- *to help ensure changes are managed and controlled*

6. multi-tiered **testing**

- *to help ensure effective error detection*

WITH SQA, IT'S PRIVILEGED TO GET OFF

cost to find
and fix a defect



SOFTWARE QUALITY

- Software quality is not just about meeting specifications and removing defects!

What is software quality and how do we measure it?

customer's viewpoint ® meets specifications

developer's viewpoint ® easy to maintain, test, . . .

Other attributes of software that affect its quality:

- safety – understandability – portability
- security – testability – usability
- reliability – adaptability – reusability
- resilience – modularity – efficiency
- robustness – complexity – learnability

We need to select critical quality attributes early in the development process and plan for how to achieve them.

PRINCIPLES OF SOFTWARE QUALITY ASSURANCE

1. We have a set of **standards** and **quality attributes** that a software product must meet.

- There is a **goal** to achieve.

2. We can **measure the quality** of a software product.



- There is a way to determine how well the product **conforms** to the standards and the quality attributes.

3. We **track** the values of the quality attributes.

- It is possible to **assess** how well we are doing.

4. We use information about software quality to **improve the quality** of future software products.

- There is **feedback** into the software development process.

SOFTWARE STANDARDS

Why are software standards important?

1. encapsulate best (or most appropriate) practices
 - acquired after much trial and error ® helps avoid previous mistakes
 2. provide a framework around which to implement SQA process
 - ensures that best practices are properly followed
 3. assist in ensuring continuity of project work
 - reduces learning effort when starting new work
- **product standards:** define the characteristics all product artifacts should exhibit so as to have quality
 - **process standards:** define how the software process should be conducted to ensure quality software

Standards Handbook needed!

Each project needs to decide which standards should be:
ignored; used as is; modified; created.

SOFTWARE METRICS

metric: any type of measurement that relates to a software system, process or related artifact

control metrics - used to plan, manage and control the development process

(e.g., effort expended, elapsed time, disk usage, etc.)

predictor metrics - used to predict an associated product quality

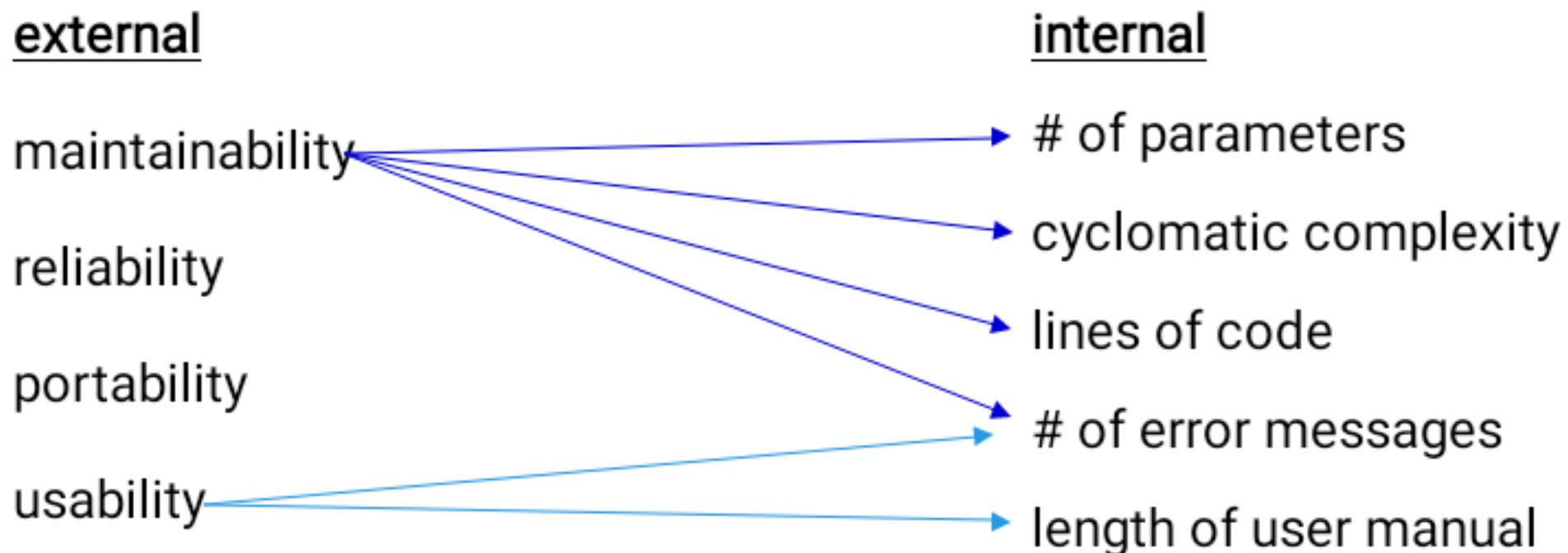
(e.g., cyclomatic complexity can predict ease of maintenance)

external attribute: something we can only discover after the software has been put into use (e.g., ease of maintenance)

internal attribute: something we can measure directly from the software itself (e.g., cyclomatic complexity)

SOFTWARE METRICS (cont'd)

We want to **use internal attributes** to predict
the value of **external attributes**.



- Problems:**
1. **hard to formulate and validate relationships** between internal and external attributes
 2. **software metrics must be collected, calibrated, interpreted**

PRODUCT QUALITY: DESIGN QUALITY METRICS

For a design component, the key quality attribute is maintainability.

For design components maintainability is related to:

- cohesion - how closely related is the functionality of the component?
- coupling - how independent is the component?
- understandability - how easy is it to understand what the component does?
- adaptability - how easy is it to change the component?

Problem: most of these cannot be measured directly, but it is reasonable to infer that there is a relationship between these attributes and the “complexity” of a component



measure complexity

How?

PRODUCT QUALITY: DESIGN QUALITY METRICS

a) Structural fan-in/fan-out

fan-in – number of calls to a component by other components

fan-out – number of components called by a component

- high fan-in => high coupling
- high fan-out => calling component has high complexity

b) Informational fan-in/fan-out

- consider also the number of parameters passed plus access to shared data structures

$$\text{complexity} = \text{component-length} \times (\text{fan-in} \times \text{fan-out})^2$$

- It has been validated using the Unix system
- It is a useful predictor of effort required for implementation

PRODUCT QUALITY: PROGRAM QUALITY METRICS

b) McCabe's Complexity Metric

Looks at **control flow** in a component.

Cyclomatic Complexity® measures component's **logical** complexity

- an indication of the **testing difficulty** of a component

Studies have shown a **distinct relationship** between the **Cyclomatic Complexity** of a component and the **number of errors** found in the source code, as well as the **time required to find and fix errors**.

c) Other program quality metrics

- length of code – length of identifiers
- depth of conditional nesting

Standards can be established to **avoid** complex components and/or **highlight** problem components.

PRODUCT QUALITY: FORMAL APPROACHES

a) Proving programs/specifications correct

- logically prove that requirements have been correctly transformed into programs

(e.g., prove assertions about programs)

b) Statistical Quality Assurance

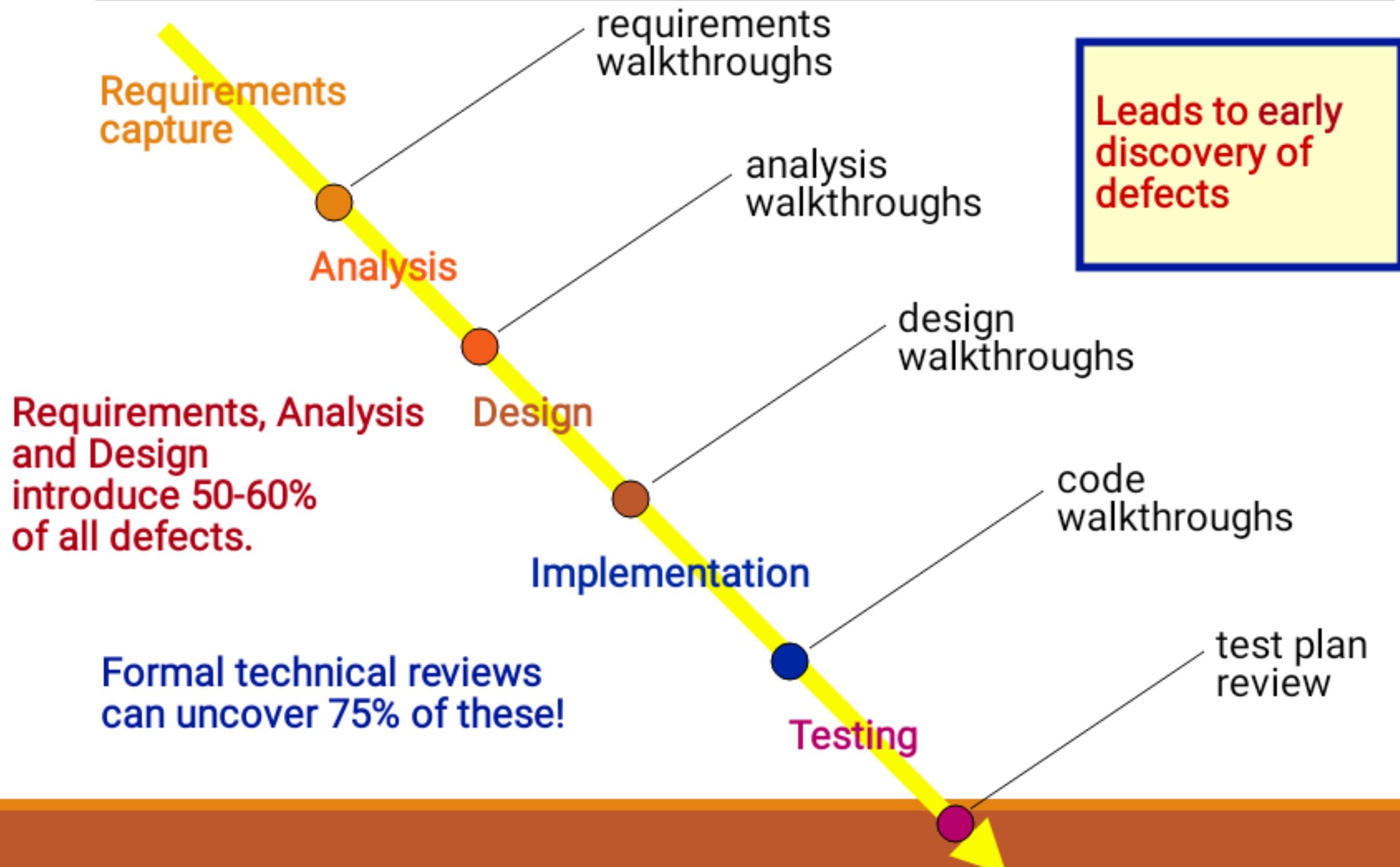
- categorize and determine cause of software defects
- **80-20 rule** ☐ 80% of defects can be traced to 20% of causes
- isolate and correct 20% of the causes
- effort directed to things that cause the majority of defects

c) The Cleanroom Process

- combination of above two approaches

PROJECT QUALITY:

The principal method of validating the quality of a project.



SOFTWARE CONFIGURATION MANAGEMENT (SCM)

An activity executed throughout the system life cycle to control change of products and life cycle artifacts.

To what do we want to control changes?

Software Configuration Management

- plans • programs
- specifications • documents/manuals
- procedures • data



configuration item: an artifact to which we want to control changes

IDENTIFICATION AND DESCRIPTION

- each configuration item must be identified and described
- a unique name
- configuration item type
- project identifier
- change and/or version information
- resources the configuration item provides or requires
- pointer to actual configuration item
- the configuration items also must be organized (i.e., need to define the
 - Define and construct a software library (database) that stores, manages and tracks configuration items.
- domain model <related-to> use-case model

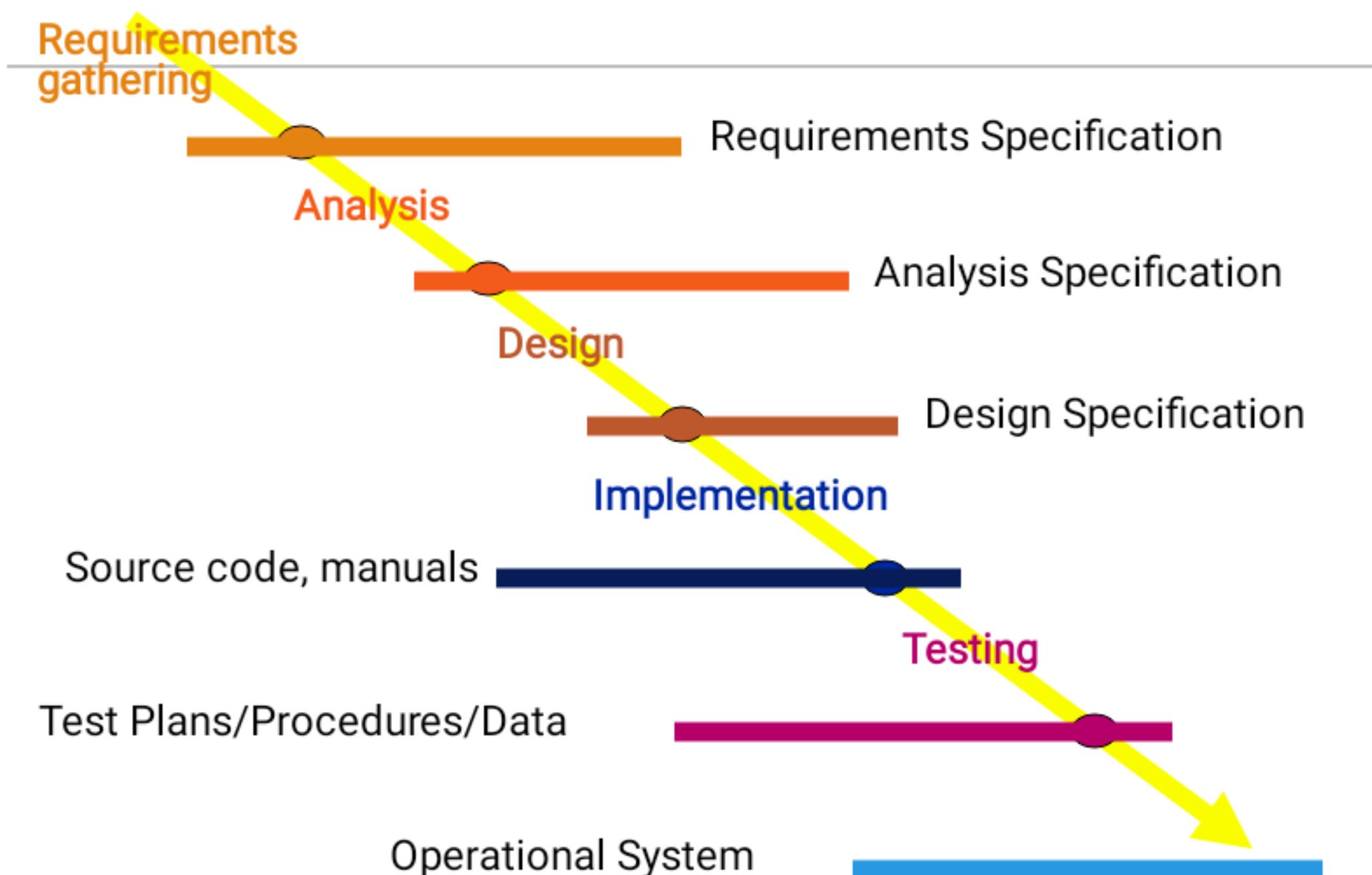
This is metadata about configuration items

CONTROLLING CONFIGURATION ITEMS: BASELINES

A baseline is a time/phase in the software development (usually a project milestone) after which any changes must be formalized (e.g., go through a formal change control procedure).

- In order to become a baseline, the configuration item must first pass a set of formal review procedures (e.g., formal code review, documentation review, etc.).
- It then becomes part of the project software library.
- After this a “check-out” procedure is applied to the item (i.e., access to and change of the configuration item is controlled)
- Any modified configuration item must again go through a formal review process before it can replace the original (baselined) item.

SCM BASELINES



CONTROLLING CONFIGURATION ITEMS: VERSION CONTROL

- a configuration item usually **evolves** throughout the software engineering process (i.e., it will have several **versions**)
- An **evolution graph** can be used to describe a configuration item's change history

version

- configuration item_k is obtained by **modifying** configuration item_i
- usually a result of bug fixes, enhanced system functionality, etc.
- item_k **supercedes original** item_i; **created in a linear order**

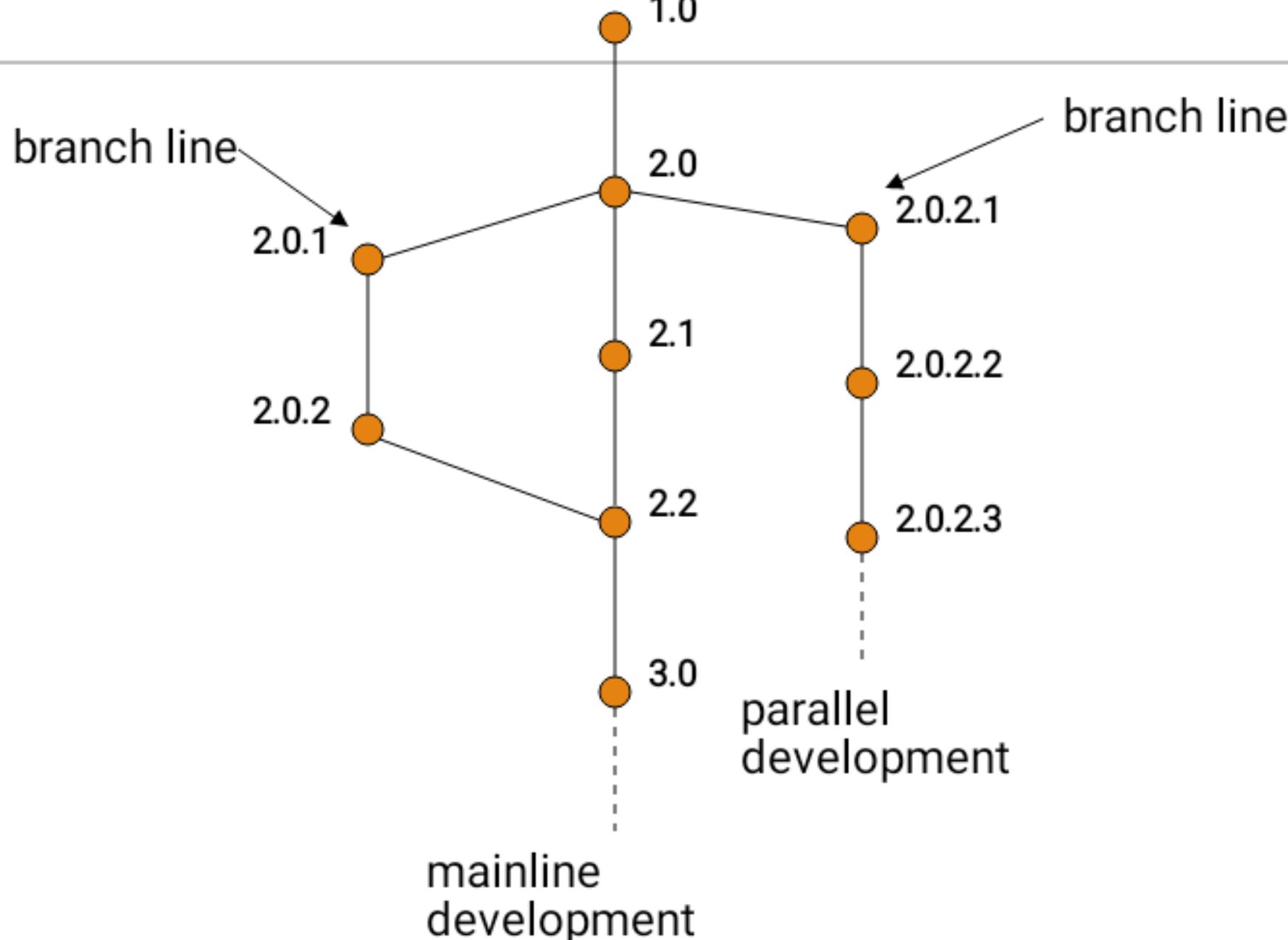
branch

- a **concurrent development path** requiring independent configuration management

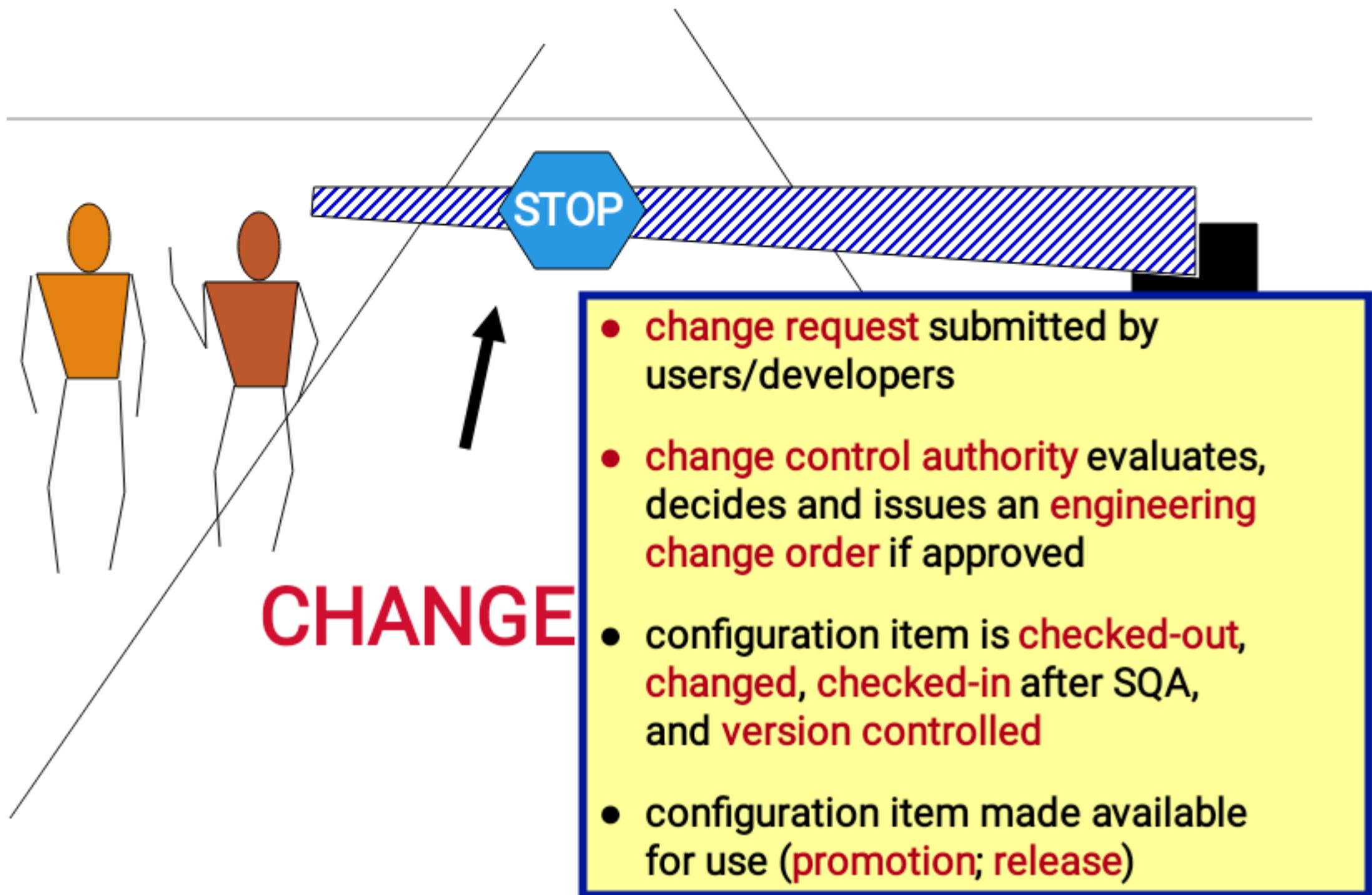
variant

- **different configurations** that are intended to **coexist**
- e.g., different configurations depending on operating system type
Oracle for Windows Oracle for Linux

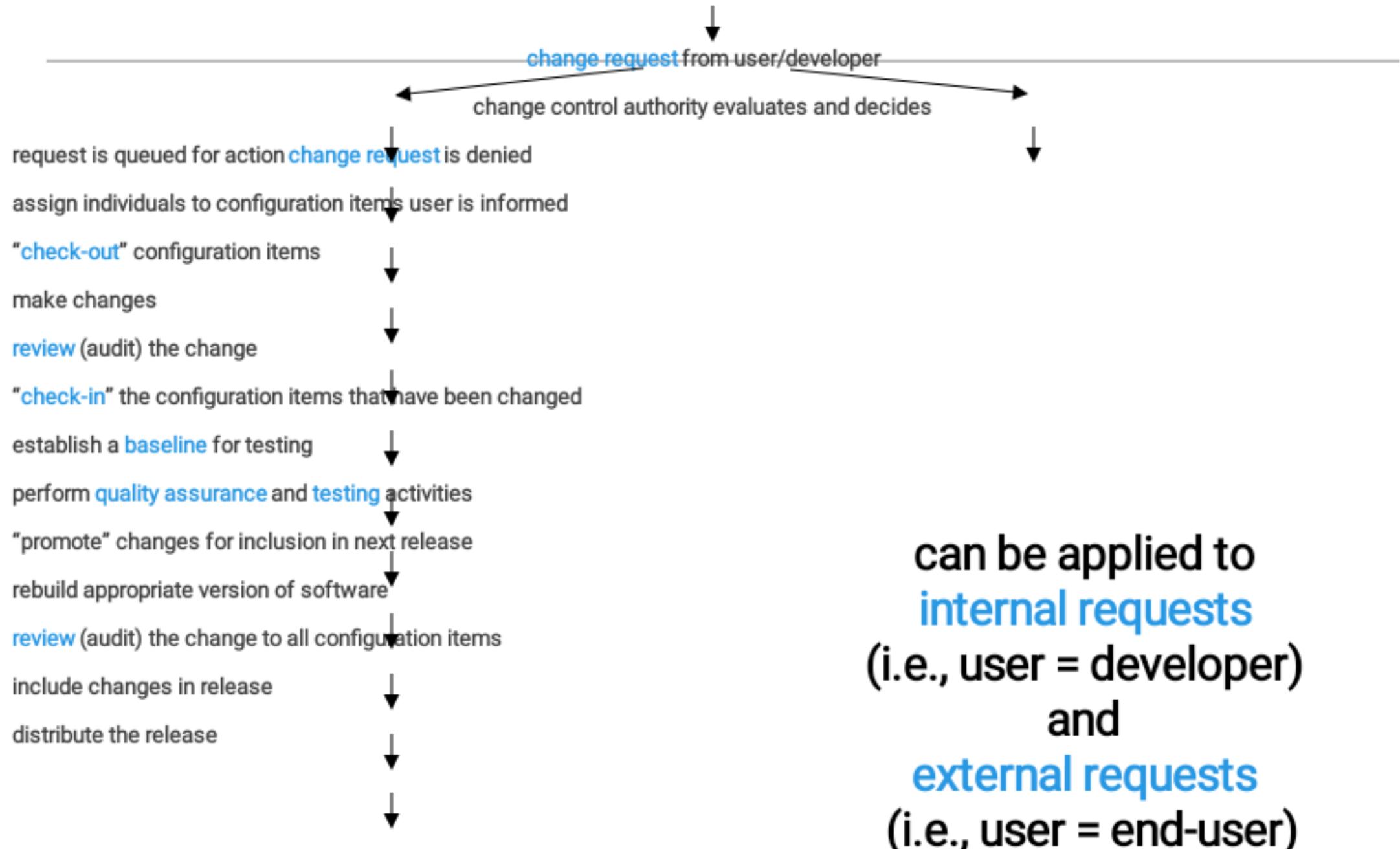
CONTROLLING CONFIGURATION ITEMS: **VERSION CONTROL**



SCM CHANGE CONTROL

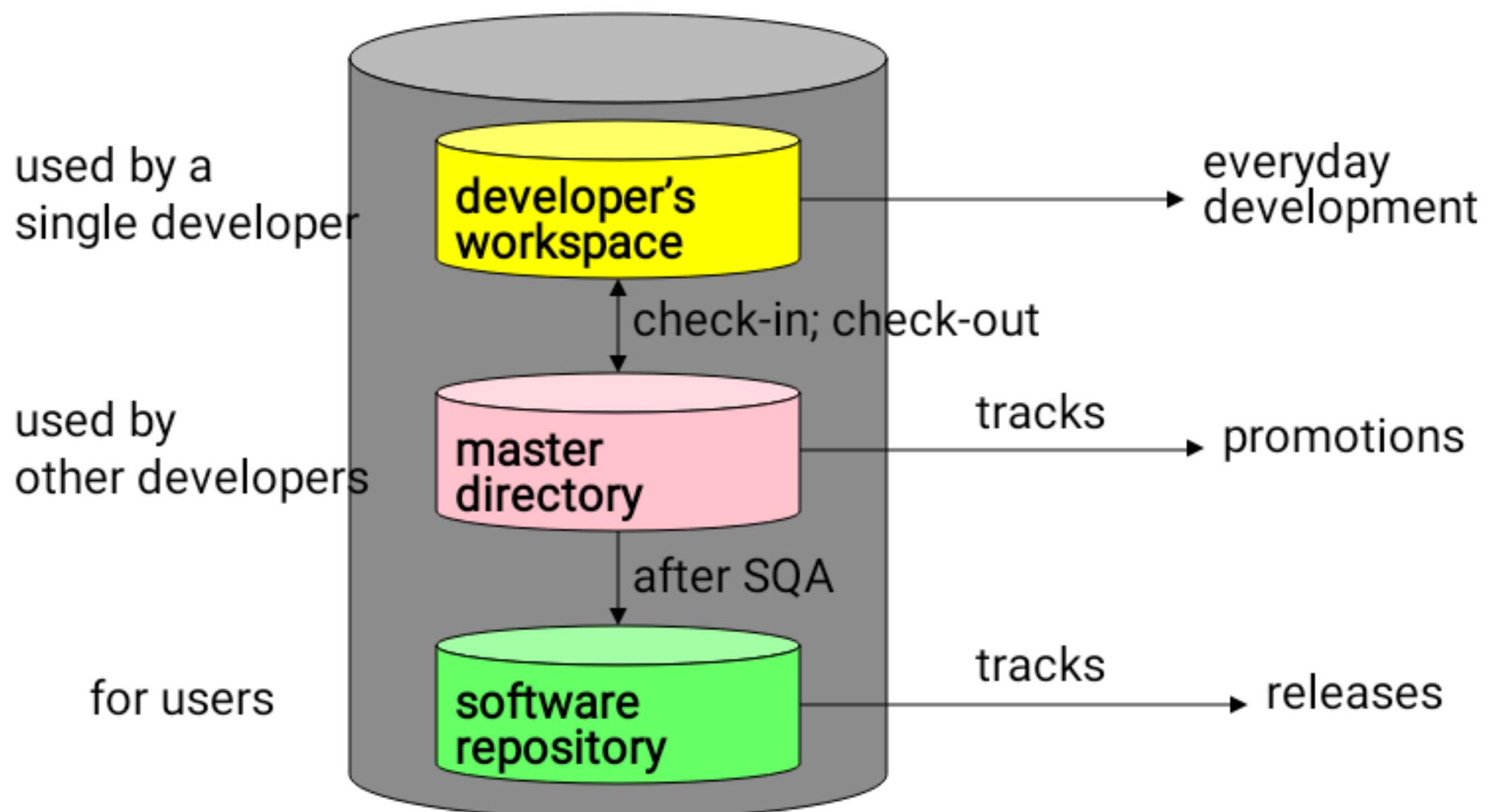


SCM CHANGE CONTROL



SCM SUPPORT

- a **software library** provides facilities to store, label, identify versions and track the status of the configuration items



SCM BENEFITS

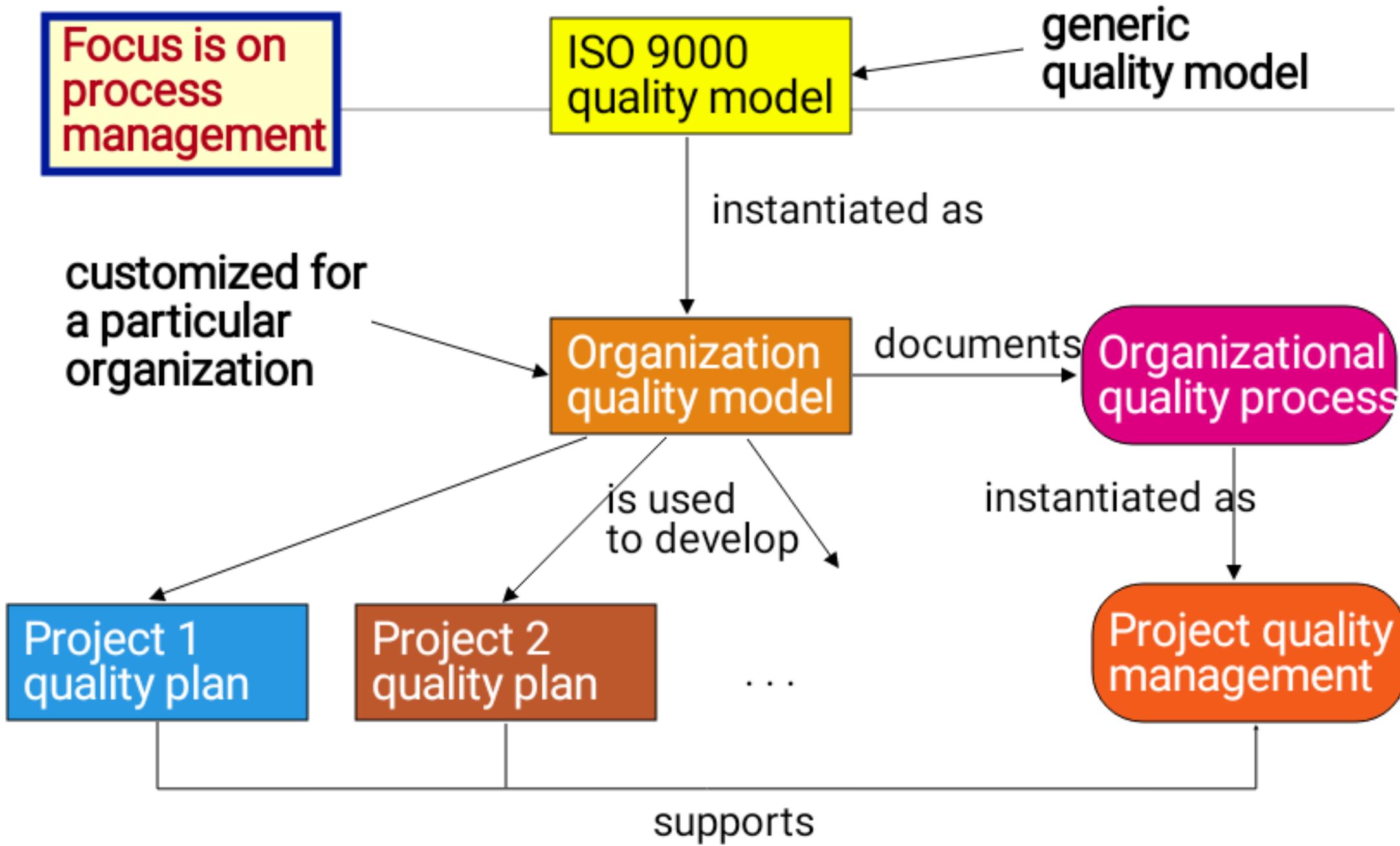
- reduces the effort required to manage and effect change
 - improved productivity
- leads to better software integrity and security
 - increased quality
- generates information about the process
 - enhanced management control
- maintains a software development database
 - better record keeping and tracking

PROCESS QUALITY

Does process quality → product quality?

- unlike manufactured products, software development has some unique factors that affect its quality:
 - software is **designed**, not manufactured
 - software development is **creative**, not mechanical
 - individual **skills** and **experience** have a **significant influence**
 - external factors (application novelty, commercial pressure)
- **software development processes are organization specific**
- **people, technology** may be more important than process
- **insufficient resources** will always adversely affect quality

PROCESS QUALITY: ISO 9001/9000-3



- certification is easy; can be a **marketing ploy**

ISO 9001/9000-3 STANDARD (1/3)

A. Quality system framework

1. Management responsibility

- quality policy defined, documented, understood, implemented and maintained
- responsibilities and authorities for all personnel defined
- in-house verification resources defined, trained and funded
- a designated manager ensures that the quality program is implemented and maintained

2. Quality system

- requires a documented quality system be established
- should be an integral process throughout the entire life cycle

3. Internal quality system audits

- requires audits be planned and performed
- results communicated to management
- deficiencies corrected

ISO 9001/9000-3 STANDARD (2/3)

B. Quality Life Cycle Activities

1. Contract review

- contracts are reviewed to determine whether the requirements are adequately defined, agree with the bid, and can be implemented

2. Purchaser's requirements specification

- procedures to identify and collect purchaser's requirements

3. Development planning

- production processes are defined and planned

4. Quality record

- quality assurance process is planned

5. Design and implementation

- includes planning design activities, identifying inputs and outputs, verifying the design and controlling design changes

6. Testing and validation

- requires systems/products to be tested and validated before use
- records of testing are kept

7. Acceptance

- procedures for acceptance should be established

8. Replication, delivery and installation

- procedures for the above should be established and maintained

9. Maintenance

- requires that maintenance activities be performed as specified

ISO 9001/9000-3 STANDARD (3/3)

C. Quality System Supporting Activities

- 1. Configuration management
 - process of development and maintenance should be documented and controlled
- 2. Document control
 - distribution and modification of documents should be controlled
- 3. Quality records
 - quality records should be collected, maintained and dispositioned
- 4. Measurement
 - where appropriate, adequate statistical techniques should be identified and used to verify the acceptability of process capability and product characteristics
- 5. Rules, practices and conventions
 - are in place and followed

- 6. Tools and Techniques
 - are applied appropriately to support the development process
- 7. Purchasing
 - purchased products conform to their specified requirements
- 8. Included software product
 - should be verified and maintained
- 9. Training
 - training needs should be identified and training provided since related tasks may require qualified personnel
 - training records should be maintained

PROCESS QUALITY: THE SEI CAPABILITY MATURITY MODEL (CMM)

Intended to help a software organization
improve their software development processes.

Level 1 Organization: Initial process (ad hoc)

- no formal procedures, no cost estimates, no project plans, no management mechanism to ensure procedures are followed

Level 2 Organization: Repeatable process (intuitive)

- basic project controls; intuitive methods used

Level 3 Organization: Defined process (qualitative)

- development process defined and institutionalized

Focus is on
process
improvement

Level 4 Organization: Managed process (quantitative)

- measured process; process database established

Level 5 Organization: Optimizing process

- improvement feedback; rigorous defect-cause analysis and prevention

CAPABILITY MATURITY MODEL (CMM)

Key Processes in Place

Level 1: Initial process

Level 2: Repeatable process

Requirements Management

Software Project Planning

Software Project Tracking & Oversight

Software Subcontract Management

Software Quality Assurance

Software Configuration Management

Level 3: Defined process

Organization Process Focus

Organization Process Definition

Level 4: Managed process

Quantitative Process Management
Software Quality Management

Level 5: Optimizing process

Fault Prevention
Technology Change Management
Process Change Management

PROCESS QUALITY – NOTES

- It is possible for a Level 1 organization to receive ISO 9000 certification!
- A Level 3 organization would have little difficulty in obtaining ISO 9000 certification.
- A Level 2 organization would have significant advantage in obtaining ISO 9000 certification.
- excellent software developers (e.g., space shuttle) attain around level 3-4 only

PEOPLE QUALITY: PEOPLE CAPABILITY MATURITY MODEL (PCMM)

Intended to improve knowledge and skill of people.

Level 1 – Initial

- no technical or management training provided; staff talent not a critical resource; no organizational loyalty

Focus is on
people
improvement

Level 2 – Repeatable

- focus on developing basic work practices; staff recruiting, growth and development important; training to fill skill “gaps”; performance evaluated

Level 3 – Defined

- focus on tailoring work practices to organization’s business; strategic plan to locate and develop required talent; skills-based compensation

Level 4 – Managed

- focus on increasing competence in critical skills; mentoring; team-building; quantitative competence goals; evaluation of effectiveness of work practices

Level 5 – Optimizing

- focus on improving team and individual skills; use of best practices

- an organization should have a **quality manual** which documents its quality assurance procedures

SQA – THE BOTTOM LINE

- each project should have a **quality plan** which sets out the **quality attributes** that are most important for that project and how they will be assessed
- an organization should have established **standards for documentation**
- mechanisms (processes) should be established that **monitor compliance** with all quality requirements of the organization
- **reviews** are the primary means of carrying out quality assurance
- **metrics** can be used to highlight anomalous parts of the software which may have quality problems

SUMMARY – SOFTWARE QUALITY

Quality software does not just happen!

- Quality assurance mechanisms should be built into the software development process
- Developing quality software requires:
 - Management support and involvement
 - Gathering and use of software metrics
 - Policies and procedures that everyone follows
 - Commitment to following the policies and procedures even when things get rough!

Testing is an important part of quality assurance, but its not all there is to obtaining a quality software product.

Syllabi for CT-2

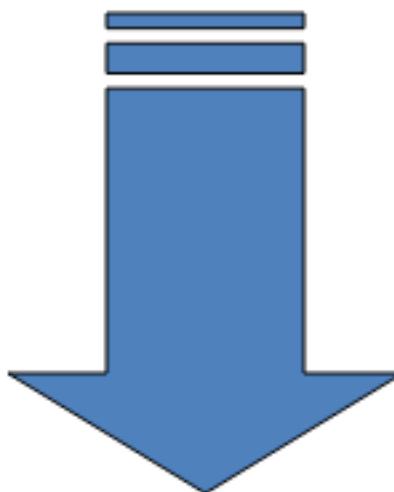
Entity Relationship Diagram, Webapp design (Slides?), Coding – Implementation Issues, Quality concepts, S/w Quality metrics – Halstead Software Science, Albrecht's Function Point Model, COCOMO Model, Cyclomatic Complexity, Verification and Validation, Software Quality Assurance

- CT-1 Toppers
- Question Pattern
- Preparation Tips – focus on concept and logic, no need to mug up except technical terms and names.
- **Book** – primary source of consultation
- Answering Tips, bring calculator
- Absenteeism in exam
- Project Question details posted along with attendance criteria – No mid sem review due to lack of time.

Verification and Validation

Slides adapted from various web sources with grateful acknowledgement of the many others who made their course materials freely available online.

Verification and Validation



**Assuring that a software
system meets a user's needs**

Objectives

- To introduce software verification and validation and to discuss the distinction between them
- To describe the program inspection process and its role in V & V
- To explain static analysis as a verification technique
- To describe the *Cleanroom* software development process

Verification vs validation

- **Verification:**

"Are we building the product right"

- The software should conform to its specification

- **Validation:**

"Are we building the right product"

- The software should do what the user really requires

The V & V process

- As a whole life-cycle process - V & V must be applied at each stage in the software process.
- Has two principal objectives
 - The discovery of defects in a system
 - The assessment of whether or not the system is usable in an operational situation.

Static and dynamic verification

- ***Software inspections*** Concerned with analysis of the static system representation to discover problems (**static verification**)
 - May be supplement by tool-based document and code analysis
- ***Software testing*** Concerned with exercising and observing product behaviour (**dynamic verification**)
 - The system is executed with test data and its operational behaviour is observed

Static and dynamic V&V

Program testing

- Can reveal the presence of errors NOT their absence !!!
- A successful test is a test which discovers one or more errors
- The only validation technique for non-functional requirements
- Should be used in conjunction with static verification to provide full V&V coverage

IV & V: Independent Validation and Verification

- Can be done by another internal team or external (other company)

developer

Understands the system
but, will test "gently"
and, is driven by "delivery"

independent tester

Must learn about the system,
but, will attempt to break it
and, is driven by quality

Types of testing

- **Defect testing**

- Tests designed to discover system defects.
- A **successful defect test** is one which reveals the presence of defects in a system.

- **Statistical testing**

- tests designed to reflect the frequency of user inputs. **Used for reliability estimation.**

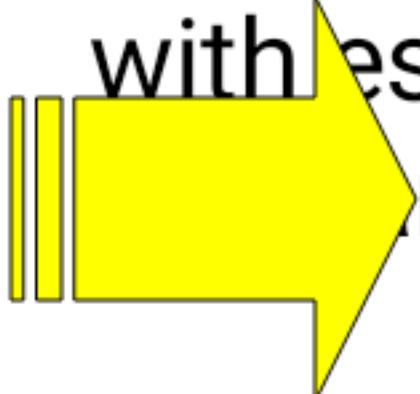
V& V goals

Verification and validation should establish confidence that the software is fit for purpose

- This does NOT mean completely free of defects

Testing and debugging

Defect testing and debugging are distinct processes

- (!) Verification and validation is concerned with establishing the existence of defects in a system
Debugging is concerned with
 - locating and
 - repairing these errors
- (!!) Debugging involves
 - formulating a hypothesis about program

The debugging process

V & V planning

- Careful planning is required to get the most out of testing and inspection processes
- Planning should start early in the development process
- The plan should identify the balance between static verification and testing
- Test planning is about defining standards for the testing process rather than describing product tests

Debugging Techniques

- ❑ brute force
- ❑ backtracking
- ❑ Cause elimination

When all else fails, ask for help!

The V-model of development

This diagram shows how test plans should be derived from the system specification and design.

The structure of a software test plan

- **The testing process** (a description of the major phases)
- **Requirements traceability** (a part of the user)
- **Tested items**
- **Testing schedule**
- **Test recording procedures** (it is not enough simply to run tests)
- **Hardware and software requirements**
- **Constraints**

Software inspections

- **Involve people** examining the source representation with the aim of discovering anomalies and defects
- **Do not require execution** of a system so may be used before implementation
- **May be applied to any representation** of the system (requirements, design, test data, etc.)
- **Very effective technique for discovering errors**

Inspection success

- Many different defects may be discovered in a single inspection. In testing, one defect, may mask another so several executions are required
- The reuse domain and programming knowledge so reviewers are likely to have seen the types of error that commonly arise

Inspections and testing

- **Inspections and testing** are complementary and not opposing verification techniques
- Both should be used during the V & V process
- Inspections can check conformance with a specification **but not conformance with the customer's real requirements**
- Also inspections cannot check non-functional characteristics such as performance, usability, etc.

Program inspections – are reviews whose objective is program defect detection.

- Formalised approach to document reviews
- Intended explicitly for defect DETECTION (not correction)
- Defects may be logical errors, anomalies in the code that might indicate an erroneous condition (e.g. an uninitialised variable) or non-compliance with standards

Inspection pre-conditions

- A precise specification must be available
- Team members must be familiar with the organisation standards
- Syntactically correct code must be available
- An error checklist should be prepared
- Management must accept that inspection will increase costs early in the software process
- Management must not use inspections for staff appraisal

The inspection process

Inspection procedure

- **System overview** presented to inspection team
 - Code and associated documents are distributed to inspection team in advance
- **Inspection** takes place and discovered errors are noted
- **Modifications** are made to repair discovered errors

Inspection teams

- Made up of at least 4 members
- **Author** of the code being inspected
- **Inspector** who finds errors, omissions and inconsistencies
- **Reader** who reads the code to the team
- **Moderator** who chairs the meeting and notes discovered errors
- Other roles are **Scribe** and **Chief moderator**

Inspection checklists

- Checklist of common errors should be used to drive the inspection
- Error checklist is programming language dependent
- The 'weaker' the type checking, the larger the checklist
- **Examples:**
 - Initialisation,
 - Constant naming,
 - loop termination,

FYI only –
not required
to know this!

Inspection checks

Inspection rate

- 500 statements/hour during overview
- 125 source statement/hour during individual preparation
- 90-125 statements/hour can be inspected
- Inspection is therefore an expensive process
- Inspecting 500 lines costs about 40 man/hours
effort = £2800

Automated static analysis

- **Static analysers** are software tools for source text processing

They parse the program text and try to discover potentially erroneous conditions and bring these to the attention of the V & V team. E.g. Embold

- Very effective as an aid to inspections. A supplement to but not a replacement for inspections

Static analysis checks

Cleanroom software development

- The name is derived from the 'Cleanroom' process in semiconductor fabrication.
- **The philosophy is defect avoidance rather than defect removal**
- Software development process based on:
 - Incremental development
 - Formal specification.
 - Static verification using correctness arguments
 - Statistical testing to determine program reliability

The Cleanroom process

Cleanroom process characteristics

- Formal specification using a state transition model
- Incremental development
- Structured programming - limited control and abstraction constructs are used
- Static verification using rigorous inspections
- Statistical testing of the system

Incremental development

Formal specification and inspections

- **The state based model** is a system specification and the inspection process checks the program against this model
- **Programming approach** is defined so that the correspondence between the model and the system is clear
- **Mathematical arguments** (not proofs) are used to increase confidence in the inspection process

Cleanroom process teams

- ***Specification team.***

- Responsible for developing and maintaining the system specification

- ***Development team.***

- Responsible for developing and verifying the software.
 - The software is NOT executed or even compiled during this process

- ***Certification team.***

- Responsible for developing a set of **statistical tests** to exercise the software after development.

Reliability growth models used to determine

Cleanroom process evaluation

- Results in IBM have been very impressive with few discovered faults in delivered systems
- Independent assessment shows that the process is no more expensive than other approaches
- Fewer errors than in a 'traditional' development process
- Not clear how this approach can be transferred to an environment with less skilled or less highly motivated engineers

Key points

- **Verification and validation** are not the same thing.
 - Verification shows conformance with specification;
 - Validation shows that the program meets the customer's needs
- **Test plans** should be drawn up to guide the testing process.
- **Static verification techniques** involve examination and analysis of the program for error detection

Key points

- **Program inspections** are very effective in discovering errors
- **Program code in inspections** is checked by a small team to locate software faults
- **Static analysis tools** can discover program anomalies which may be an indication of faults in the code
- **The Cleanroom development process** depends on
 - incremental development,
 - static verification and statistical testing

Web App design

References : Textbook(pressman 7th edition), Webpage(wikipidea).

By :

- Bhavika Mahajan**
- Dhrubit Hajong**
- Rogeshwaran Perumal**
- Simran Singh**

“There are essentially two basic approaches to design:the artistic ideal of expressing yourself and the engineering ideal of solving a problem for a customer”

-Jacob Nielsen[Nieoo]states

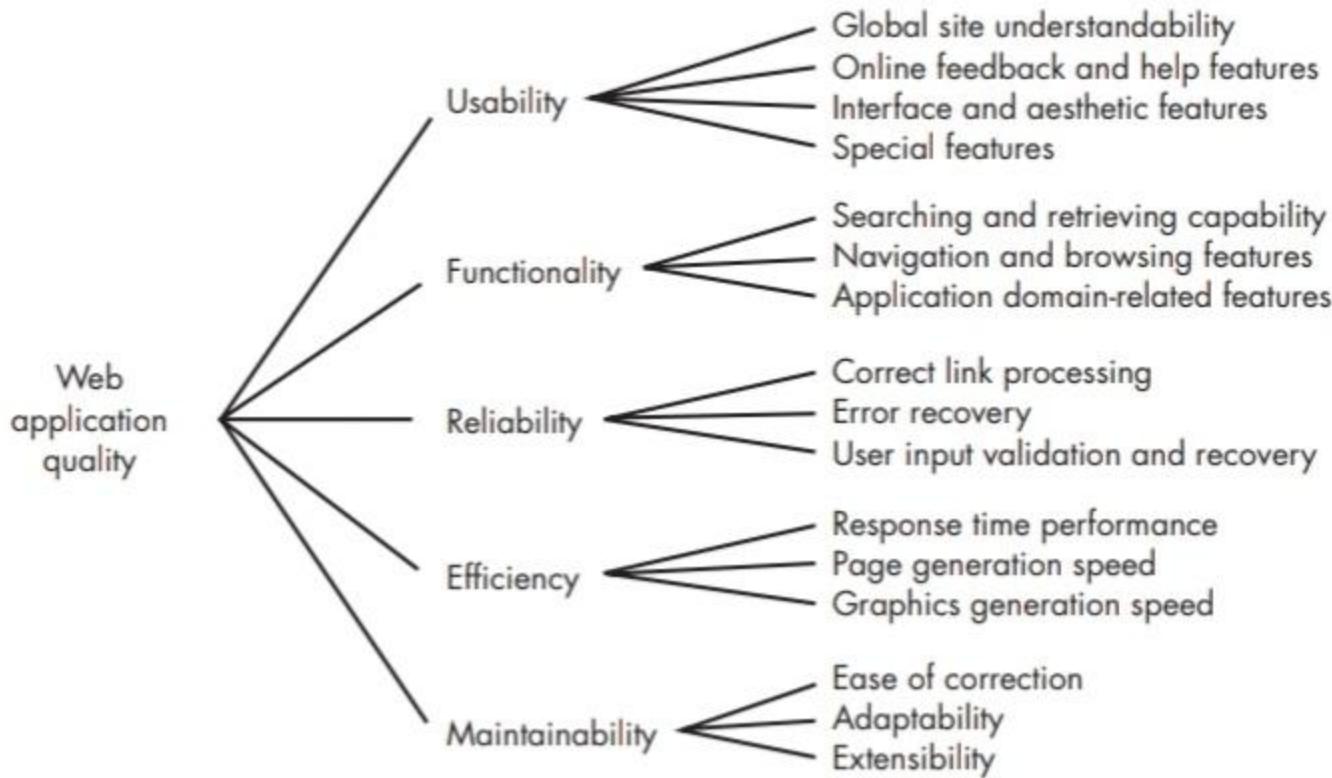
WHAT IS IT??

Design for Webapps encompasses technical and non technical activities that includes:

- ❖ Establishing the looks and feel of the WebApp.
- ❖ Creating the aesthetic layout of the user interface.
- ❖ Defining the overall architecture structure.
- ❖ Developing the content and functionality that resides within the architecture.
- ❖ Planning the navigation that occurs within the webapps.

When should we emphasize WebApp design

- ❖ When content and functions are complex.
- ❖ When the size of webapp encompasses hundreds of content objects ,functions, and analysis classes.
- ❖ When the success of webapp will have direct impact on the success of Business.



WebApp Design Quality

Security

- Rebuff external attacks
- Exclude unauthorized access
- Ensure the privacy of users/customers

Availability

- The measure of the percentage of time that a WebApp is available for use.

Scalability

- Can the WebApp and the systems with which it is interfaced handle significant variation in user or transaction volume

Quality Dimensions for end-users

■ Time

- How much has a Website changed since the last upgrade?
- How do you highlight the parts that have changed?

■ Structural

- How well do all of the parts of the Web site hold together.
- Are all links inside and outside the Web site working?
- Do all of the images work?
- Are there parts of the Web site that are not connected?

■ Content

- Does the content of critical pages match what is supposed to be there?
- Do key phrases exist continually in highly-changeable pages?
- Do critical pages maintain quality content from version to version?
- What about dynamically generated HTML pages?

■ Accuracy and Consistency

- Are today's copies of the pages downloaded the same as yesterday's?
Close enough?
- Is the data presented accurate enough? How do you know?

■ Response Time and Latency

- Does the Web site server respond to a browser request within certain parameters?
- In an E-commerce context, how is the end to end response time after a SUBMIT?
- Are there parts of a site that are so slow the user declines to continue working on it?

■ Performance

- Is the Browser-Web-Web site-Web-Browser connection quick enough?
- How does the performance vary by time of day, by load and usage?
- Is performance adequate for E-commerce applications?

WebApp Design Goals

Consistency

- **Content** should be constructed consistently
- **Graphic design** (aesthetics) should present a consistent look across all parts of the WebApp
- **Architectural design** should establish templates that lead to a consistent hypermedia structure
- **Interface design** should define consistent modes of interaction, navigation and content display
- **Navigation** mechanisms should be used consistently across all WebApp elements

Identity

- Establish an “identity” that is appropriate for the business purpose

Robustness

- The user expects robust content and functions that are relevant to the user’s needs

Navigability

- Designed in a manner that is intuitive and predictable

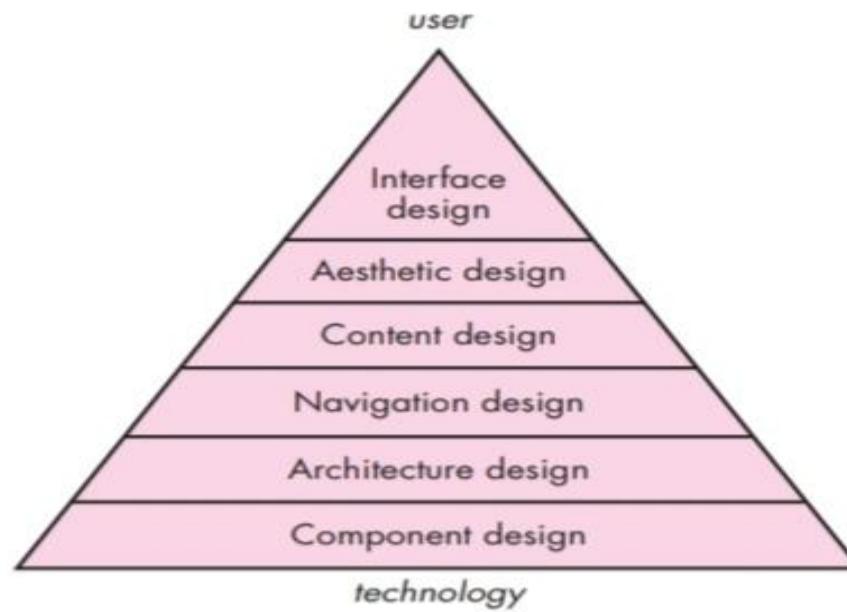
Visual appeal

- The look and feel of content, interface layout, color coordination, the balance of text, graphics and other media, navigation mechanisms must appeal to end-users

Compatibility

- With all appropriate environments and configurations

WebApp Design pyramid



WEBAPP INTERFACE DESIGN

The objective of a WebApp interface are

- ❖ provide an indication of the WebApp that has been accessed
- ❖ inform the user of her location in the content hierarchy.
- ❖ establish a consistent window into the content and functionality provided by the interface
- ❖ guide the user through a series of interactions with the WebApp
- ❖ organize the navigation options and content available to the user.
- ❖ To achieve a consistent interface, we should use aesthetic design to establish a coherent look.

Effective WebApp Interfaces

- ★ Good user interfaces provide their users a sense of control by being **intuitive and forgiving** to mistakes. Users can see the range of their selections fast. **comprehend** how to carry out their **tasks and achieve** their goals.
- ★ The user is not **bothered** by the system's **internal** workings thanks to effective interfaces. Work is **meticulously and consistently** saved, and the user always has the opportunity to go back and **undo any action**.
- ★ Applications and services that are effective accomplish the most with the least amount of **user input**.

Effective WebApp Interfaces

To gain an intuitive understanding of the interface:

To implement navigation options, we can select from a number of interaction mechanism :

- ★ **Navigation menu** : keyword menus that highlight important functionality or content. As the user selects the main menu choice, a **hierarchy of subtopics** may be displayed, allowing them to be selected from by the user.
- ★ **Graphic icons** : buttons, switches, and other similar **graphical images** that let the user **describe a choice or pick a property.**
- ★ **Graphic images** : A link to a content item or WebApp capability is implemented by some **selectable graphical representation.**

Interface Design Principles-I

- **Anticipation:** A WebApp should be designed so that it anticipates the user's next move.
- **Communication:** The interface should communicate the status of any activity initiated by the user.
- **Consistency:** The use of navigation controls, menus, icons, and aesthetics (e.g., color, shape, layout)
- **Controlled autonomy:** The interface should facilitate user movement throughout the WebApp, but it should do so in a manner that enforces navigation conventions that have been established for the application.
- **Efficiency:** The design of the WebApp and its interface should optimize the user's work efficiency, not the efficiency of the Web engineer who designs and builds it or the client-server environment that executes it.

Interface Design Principles-II

- **Focus:** The WebApp interface (and the content it presents) should stay focused on the user task(s) at hand.
- **Fitt's Law:** “The time to acquire a target is a function of the distance to and size of the target.”
- **Human interface objects:** A vast library of reusable human interface objects has been developed for WebApps.
- **Latency reduction:** The WebApp should use multi-tasking in a way that lets the user proceed with work as if the operation has been Completed.
- **Learnability:** A WebApp interface should be designed to minimize learning time, and once learned, to minimize relearning required when the WebApp is revisited.

Interface Design Principles-III

- **Maintain work product integrity:** A work product (e.g., a form completed by the user, a user specified list) must be automatically saved so that it will not be lost if an error occurs.
- **Readability:** All information presented through the interface should be readable by young and old.
- **Track state:** When appropriate, the state of the user interaction should be tracked and stored so that a user can logoff and return later to pick up where she left off.
- **Visible navigation:** A well-designed WebApp interface provides “**the illusion that users are in the same place, with the work brought to them.**”

Aesthetic Design

Why Are Aesthetics Important?

- In Conventional web design , most functions and options appeared randomly, without interacting with the user.
- The general user had to work harder to understand their navigation patterns due to the complex functionality of the technology`s back end.
- Modern web design solves these problems by providing new coding techniques, better tools, and the latest software development models.

- Web design should be clean in appearance and developed using Human-Computer Interaction Techniques to gain the audience's attention and provide them with value.
- Aesthetics is critical in online marketplaces and e-commerce platforms by focusing on the target audience.
- Websites with multiple components separate product categories instead of bombarding users with different products on the screen.
- Similarly, different website products are marketed via images and short videos to increase their visual appeal.

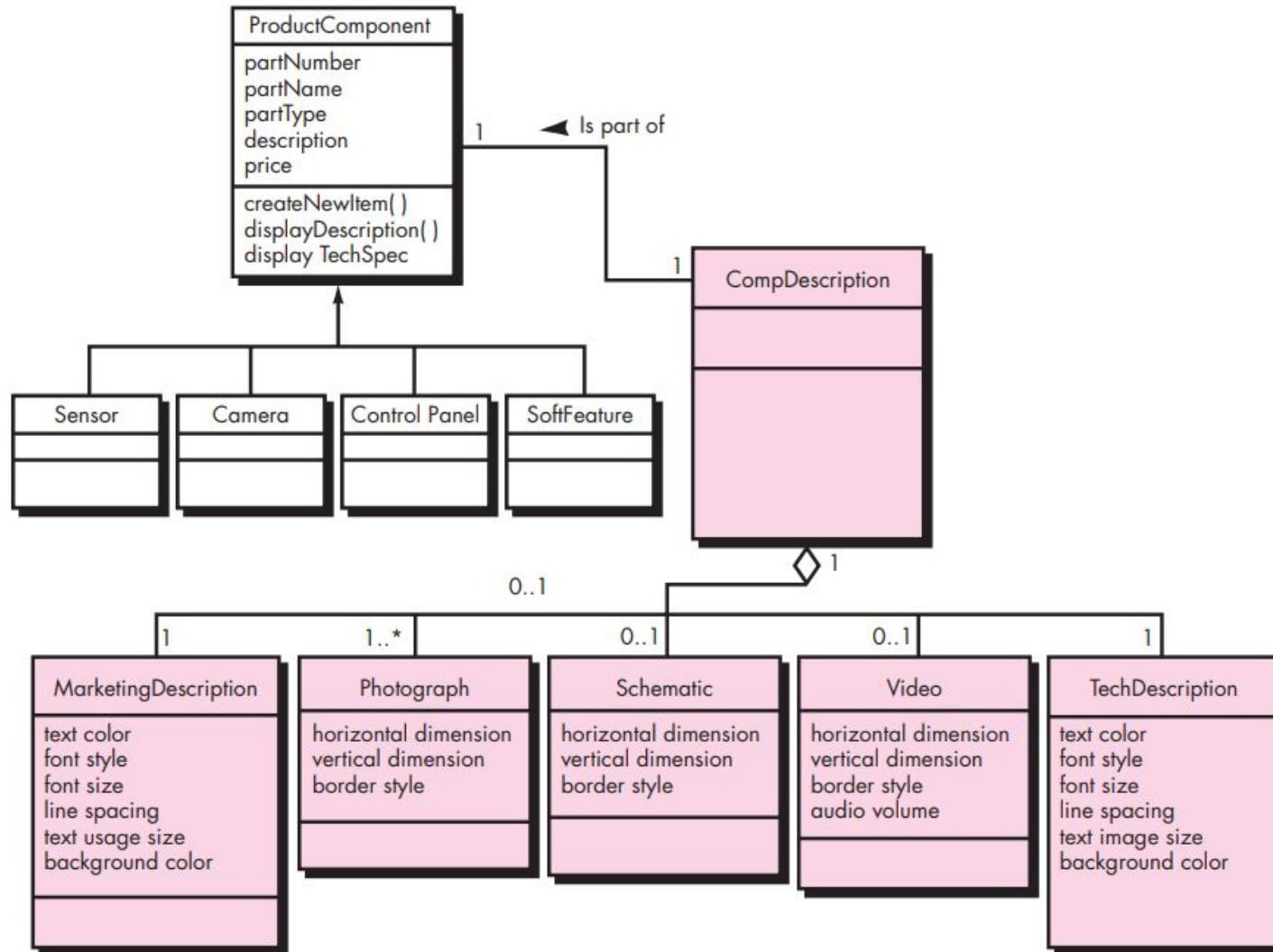
General layout guidelines for Aesthetic Design

- **Do not be afraid of white space.** It is inadvisable to pack every square inch of a Web page with information. The resulting clutter makes it difficult for the user to identify needed information or features and creates visual chaos that could be more pleasing to the eye.
- **Emphasize content.** After all, that is the reason the user is there. A typical Web page should be 80 percent contained, with the remaining real estate dedicated to navigation and other features.
- **Organize layout elements from top-left to bottom-right:** Most users scan a Web page in much the same way as they scan the page of a book—top-left to bottom-right. If layout elements have specific priorities, high-priority elements should be placed in the upper-left portion of the page real estate.

- **Group navigation, content, and function geographically within the page:** Humans look for patterns in virtually all things. If there are no patterns within a Web page, user frustration is likely to increase.
- **Do not extend your real estate with the scrolling bar:** Although scrolling is often necessary, most studies indicate that users prefer not to scroll. Reducing page content or presenting actual content on multiple pages is better.

CONTENT DESIGN

- Content design focuses on two different design tasks, each addressed by individuals with different skill sets. First, a design representation for content objects and the mechanisms required to establish their relationship to one another is developed. In addition, the information within a specific content object is created.



Content Design Issues

- Once all content objects are modeled, each piece of information must be authored and formatted to meet the customer's needs best. Content authoring is the job of specialists in the relevant area who design the content object by providing an outline of information to be delivered, and an indication of the types of generic content objects (e.g., descriptive text, graphic images, photographs) used to deliver the information.

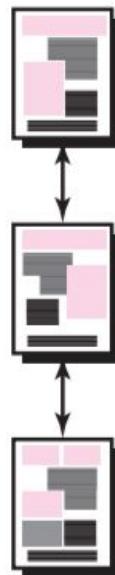
Architecture design

- Architecture design is tied to the goals established for a WebApp, the content to be presented, the users who will visit, and the navigation philosophy that has been established. WebApp architecture addresses how the application is structured to manage user interaction, handle internal processing tasks, affect navigation, and present content. In most cases, architecture design is conducted parallel with interface design, aesthetic design, and content design. Because the WebApp architecture may strongly influence navigation, the decisions made during this design action will influence work conducted during navigation design

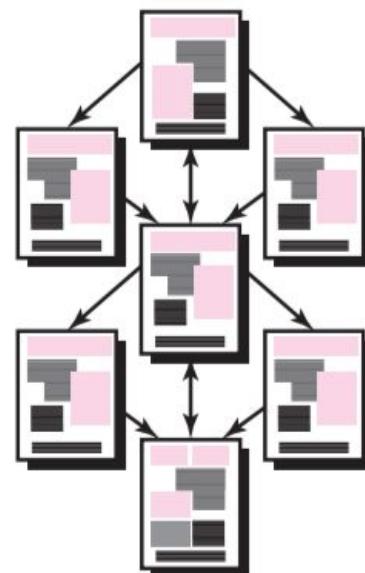
Content Architecture

- The design of content architecture focuses on the definition of the overall hypermedia structure of the WebApp. Although custom architectures are sometimes created, we always have the option of choosing from four different content structures

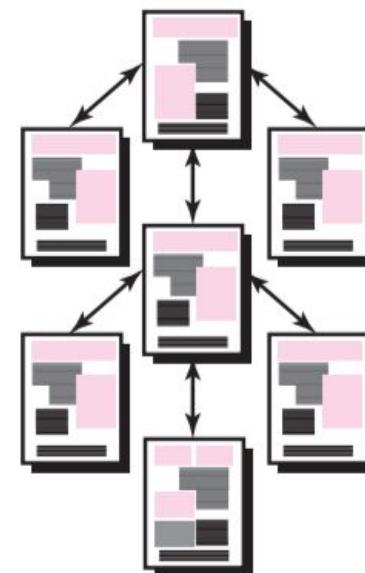
LINEAR STRUCTURES



Linear

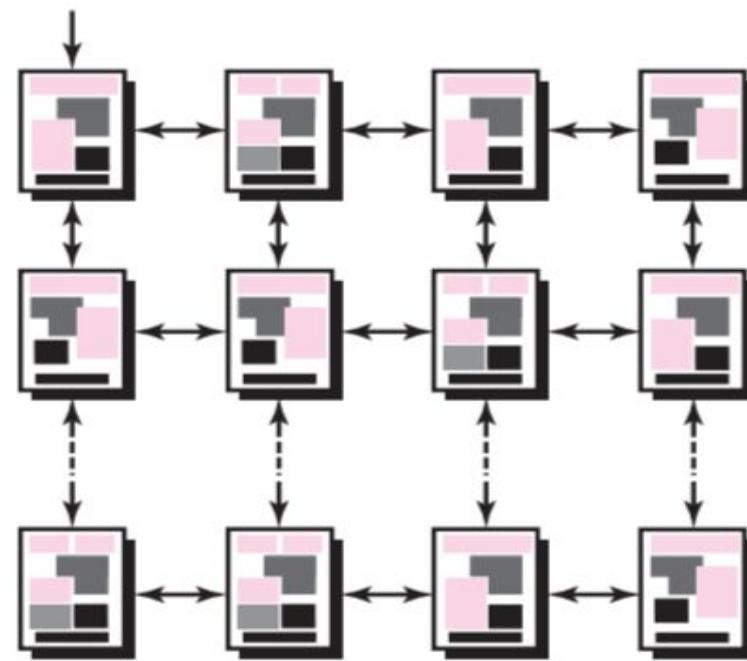


Linear
with
optional flow

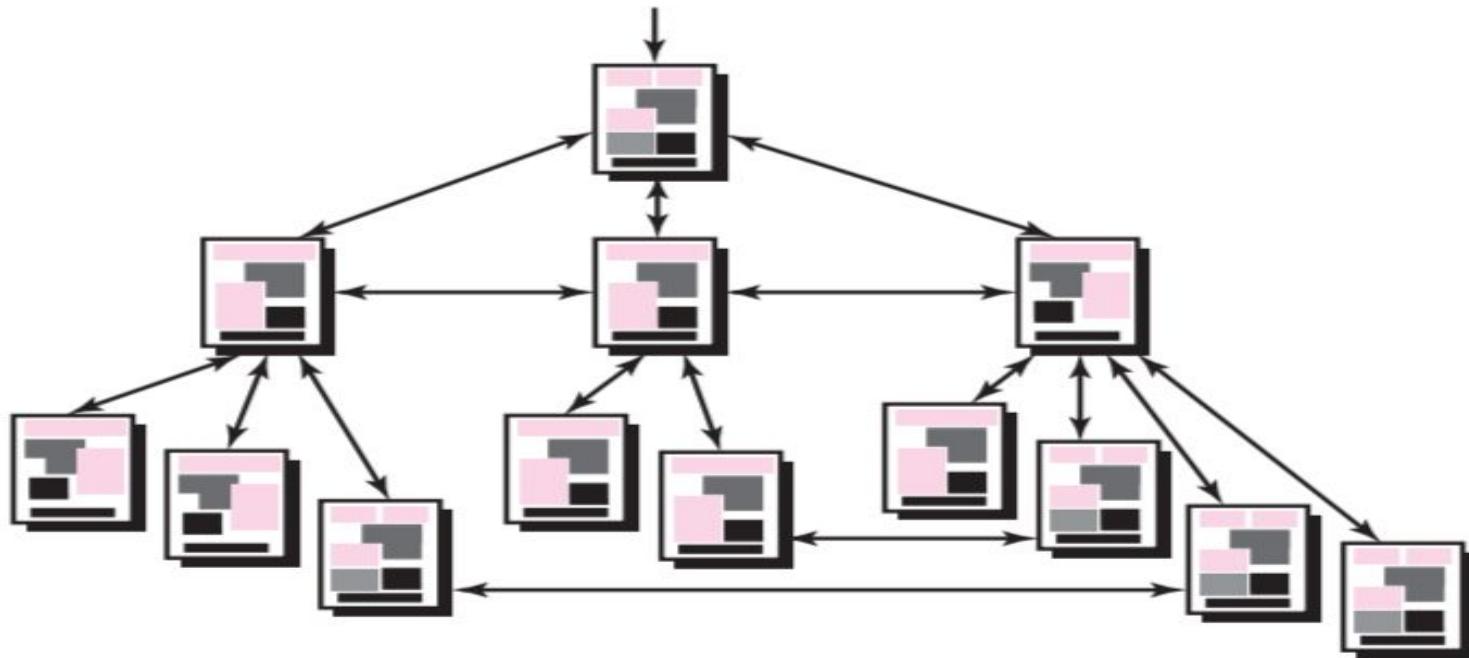


Linear
with
diversions

GRID STRUCTURES



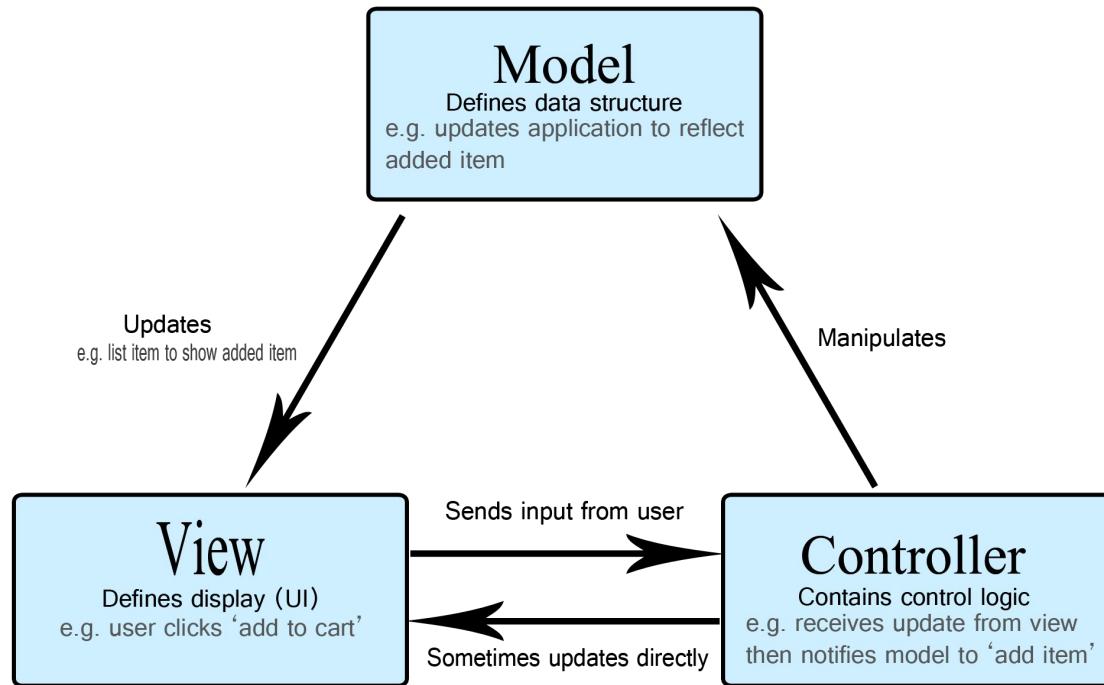
Hierarchical structure



MVC

(Model-View-Controller)

The **Model-View-Controller (MVC)** is an architectural pattern that separates an application into three main logical components: the **model**, **the view**, and **the controller**. Each component is built to handle specific development aspects of an application. MVC is one of the most frequently used industry-standard web development frameworks to create scalable and extensible projects.



Navigation Design

In short Navigation design represents various path that a user takes to accomplish their goal.

It could differ from user to user.\

- 1) Find semantics of navigation of different user.
- 2) Define the mechanics, syntax of Navigation.

Navigation Semantics

Specific use-cases need few details.

- 1) Information: Present location, destination.
- 2) Direction/Path: A single source to destination could have multiple paths.

Formal definition: A set of information and related navigation structures that collaborate in the fulfillment of a subset of related user requirements.

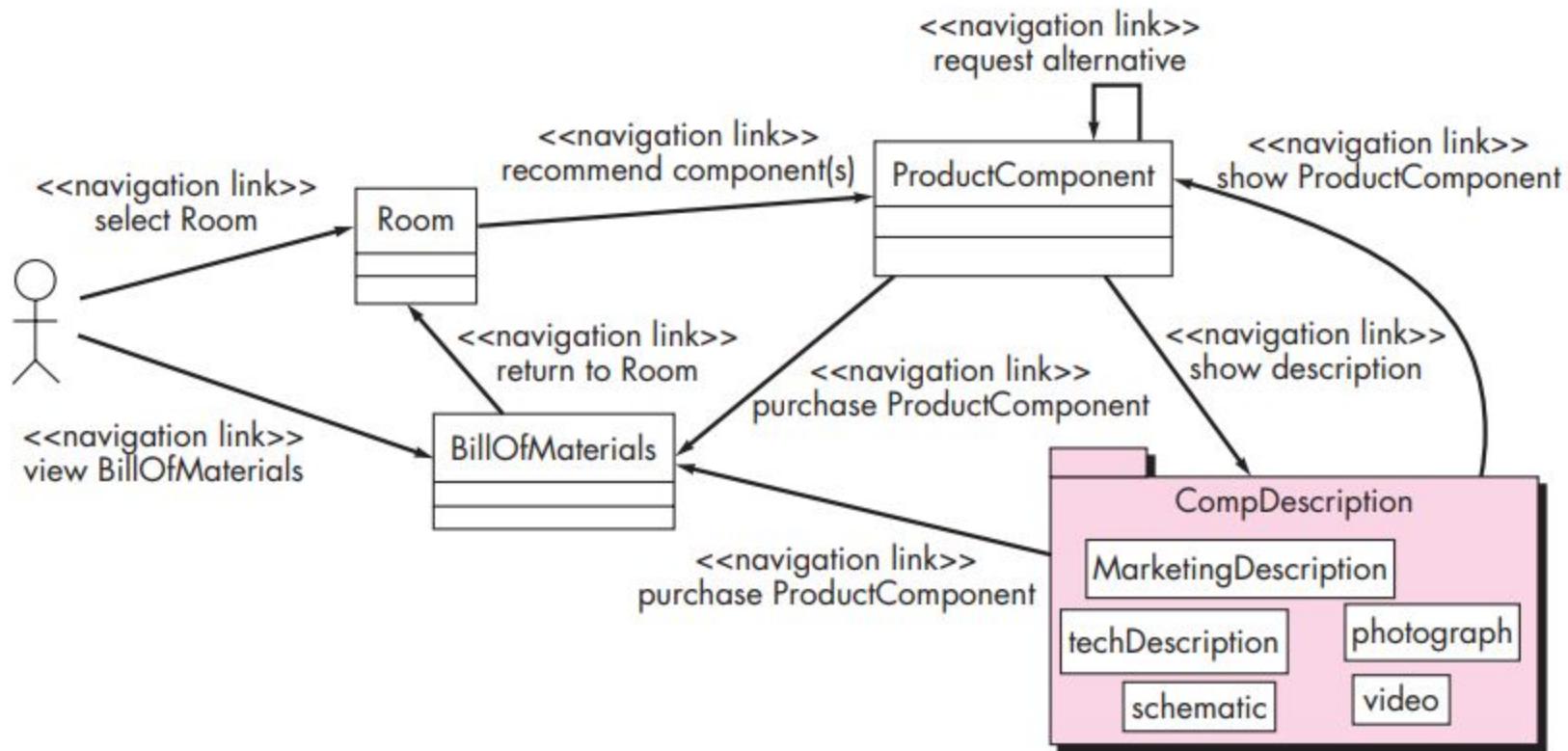
Navigation semantic units

Ways of navigation (WoN)

Represents the best navigation way or path for users with certain profiles to achieve their desired goal or sub-goal.

Use Case: Select SafeHome Components

The WebApp will recommend product components (e.g., control panels, sensors, cameras) and other features (e.g., PC-based functionality implemented in software) for each room and exterior entrance. If I request alternatives, the WebApp will provide them, if they exist. I will be able to get descriptive and pricing information for each product component. The WebApp will create and display a bill-of-materials as I select various components. I'll be able to give the bill-of-materials a name and save it for future reference (see use case **Save Configuration**).



Navigation Syntax

Individual navigation link—text-based links, icons, buttons and switches, and graphical metaphors..

Horizontal navigation bar—lists major content or functional categories in a bar containing appropriate links. In general, between 4 and 7 categories are listed.

Vertical navigation column —

- lists major content or functional categories
- lists virtually all major content objects within the WebApp

Tabs—a metaphor that is nothing more than a variation of the navigation bar or column, representing content or functional categories as tab sheets that are selected when a link is required.

Site maps—provide an all-inclusive tab of contents for navigation to all content objects and functionality contained within the WebApp.

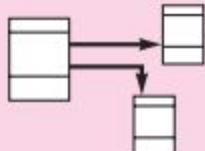
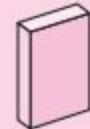
Component Design

- WebApp components implement the following functionality
- perform localized processing to generate content and navigation capability in a dynamic fashion
- provide computation or data processing capability that are appropriate for the WebApp's business domain
- provide sophisticated database query and access
- establish data interfaces with external corporate systems

Object-Oriented Hypermedia Design Method

Object oriented Design approach which consists of Four different stages:

- 1) Conceptual design
- 2) Navigational design
- 3) Abstract interface design
- 4) Implementation

				
	Conceptual design	Navigational design	Abstract interface design	Implementation
Work products	Classes, subsystems, relationships, attributes	Nodes links, access structures, navigational contexts, navigational transformations	Abstract interface objects, responses to external events, transformations	Executable WebApp
Design mechanisms	Classification, composition, aggregation, generalization specialization	Mapping between conceptual and navigation objects	Mapping between navigation and perceptible objects	Resource provided by target environment
Design concerns	Modeling semantics of the application domain	Takes into account user profile and task. Emphasis on cognitive aspects.	Modeling perceptible objects, implementing chosen metaphors. Describe interface for navigational objects.	Correctness; application performance; completeness

Conceptual Design

OOHDM conceptual design creates a representation of the subsystems, classes, and relationships that define the application domain for the WebApp.

UML may be used¹² to create appropriate class diagrams, aggregations, and composite class representations, collaboration diagrams, and other information that describes the application domain

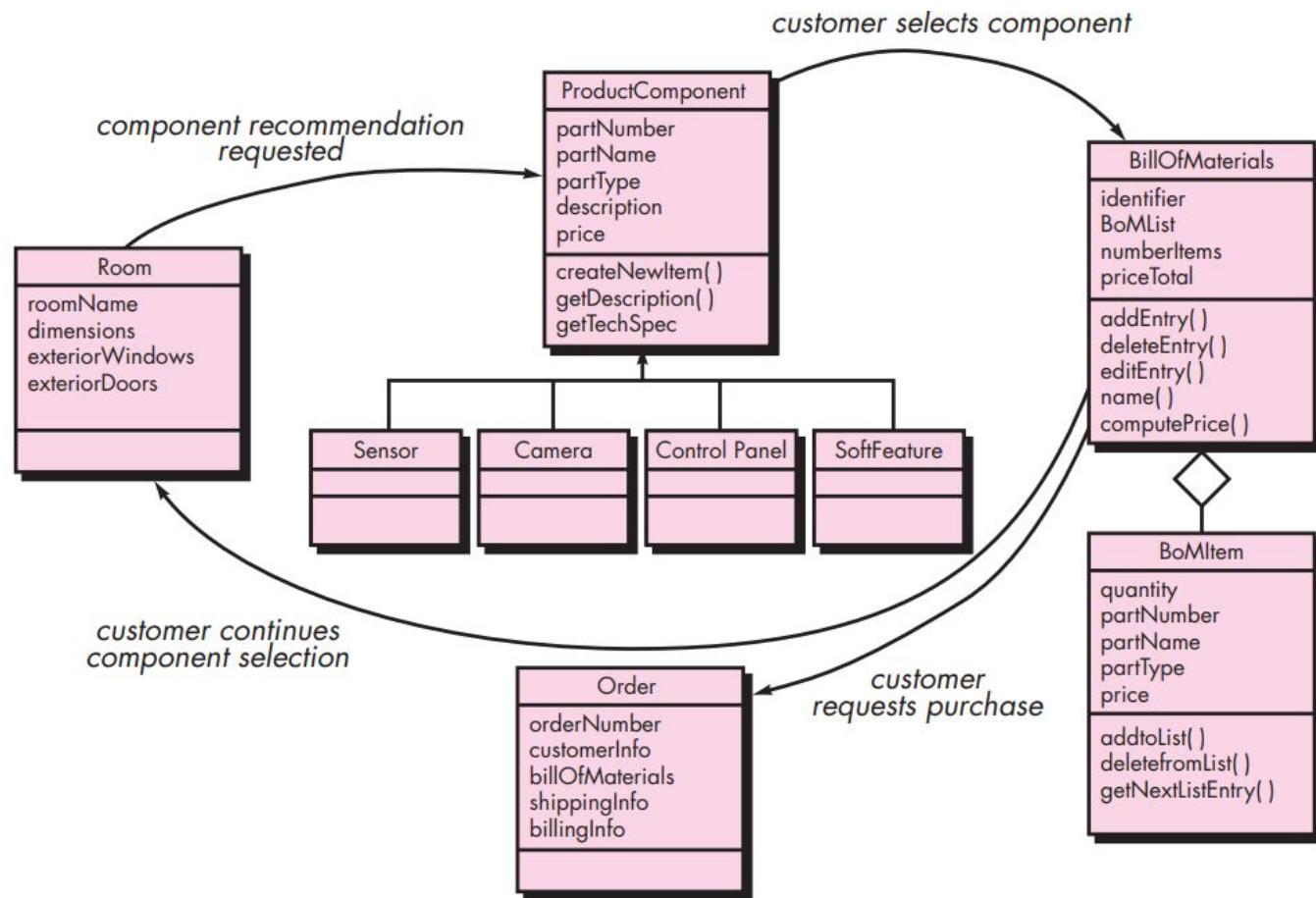
The class diagrams, aggregations, and related information developed as part of WebApp analysis are reused during conceptual design to represent relationships between classes

Navigational Design for OOHDM

Navigational design identifies a set of “objects” that are derived from the classes defined in conceptual design.

A series of “navigational classes” or “nodes” are defined to encapsulate these objects

OOHDM uses a predefined set of navigation classes—nodes, links, anchors, and access structures

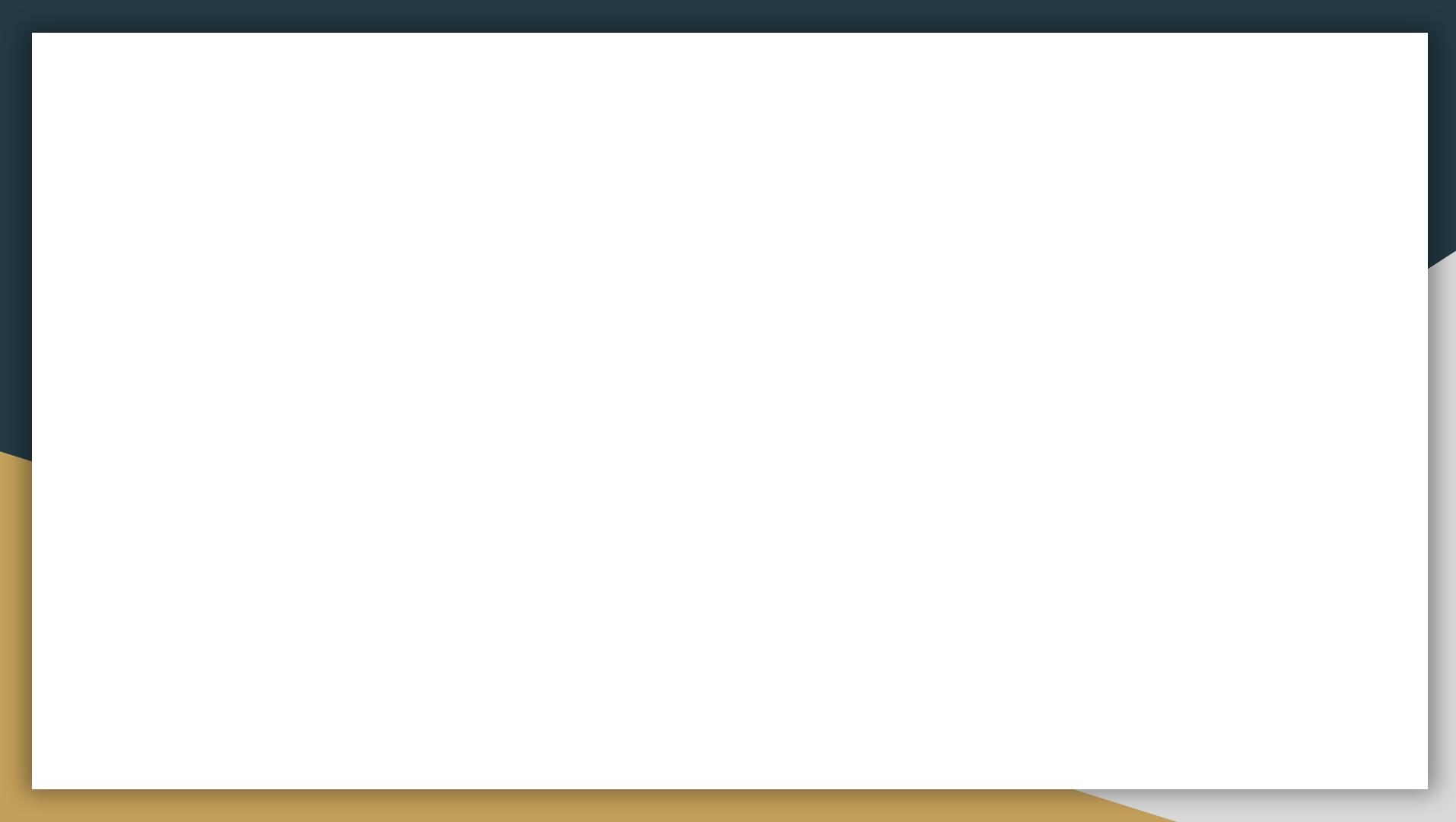


Abstract Interface Design and Implementation

The abstract interface design action specifies the interface objects that the user sees as WebApp interaction occurs. A formal model of interface objects, called an abstract data view (ADV), is used to represent the relationship between interface objects and navigation objects, and the behavioral characteristics of interface objects.

The ADV model defines a “static layout” that represents the interface metaphor and includes a representation of navigation objects within the interface and the specification of the interface objects (e.g., menus, buttons, icons) that assist in navigation and interaction.

“trigger navigation and which interface transformations occur when the user interacts with the application”



Length Measures

$$LOC = NCLOC + KCLOC$$

↓ ↓
Non commented commented

$$\text{Ratio} = \frac{CLOC}{LOC} \Rightarrow \text{Density of comments}$$

Halstead's Metrics

n_1 = # distinct operators

n_2 = # of distinct operands

N_1 = total no. of occurrences of operators

N_2 = "

Halstead: $A(I) = A(J)$

Operator: 1 (=)

Operands: 2 $A(I)$ and $A(J)$

Length of P is $N = N_1 + N_2$

Vocabulary of P is $n_1 + n_2$.

Volume of a program (# of mental comparisons)

needed to write a program of length N)

$$V = N \times \log_2(n).$$

Program level of 'P' of volume N is

$$L = V^*/V$$

V^* Potential volume - volume of the minimal size implementation of P.
(or volume of the most succinct programs which a problem can be coded)

Level of D = 1/L

Difficulty.

a problem can be coded generally $V^* = 11.6$

estimate of \hat{L} of L is

$$\hat{L} = \frac{1}{D} = \frac{2}{n_1} \times \frac{n_2}{N_2}$$

Higher the level of a language, the less effort it takes to develop a program using that language.

$$L = [0, 1]$$

$L = 1 \rightarrow$ means - a program written at the highest point level (i.e., with min. size)

An estimate of \hat{L} of L is

$$\hat{L} = \frac{1}{D} = \frac{2}{n_1} \times \frac{n_2}{N_2}$$

Estimated length is

$$\boxed{N = n_1 \times \log_2 n_1 + n_2 \times \log_2 n_2}$$

Redundant usage of Operands } Inexact Volume
Failure to use higher-level Control constructs } Difficulty

Programming

Effort: - Elementary mental discriminations needed to understand P.
[Mental activity needed to translate the existing algo. to Implementation in the specified program language].

$$E = \frac{N}{L} = \frac{N}{\frac{D}{B}} = N \times D = \frac{N_1 N_2 N \log_2 n}{2n_2}$$

A psychologist named John Shoultz -
Human mind is capable of making a limited number, B , of elementary discriminations / sec.

$$5 \leq B \leq 20$$

Halstead : $B = 18$, for a programming language & Programming Time, $T = E/18$ seconds

Operators	Ocurrences	Operands	Ocurrences
int	4	sort	1
()	5	A	7
1	4	n	3
[]	7	i	8
if	2	j	7
<	2	any	3
;	11	mn	3
for	2	2N	2
=	6	1	3
-	1	0	1
<=	2	-	-
++	2	-	-
return	2	-	-
{ }	3	-	-
<hr/>		$n_1 = 53$	$n_2 = 10$
$n_1 = 14$		$N = n_1 + n_2 = 91$	$N_2 = 38$
$m = m_1 + m_2 = 211$		$V = 91 \times \log(24)$	$V = 417.25$ bits

$$V^* = 11.6$$

$$L = \frac{11.6}{417.23} = 0.027$$

$$D = \frac{1}{L} = \frac{1}{0.027} = 37.03.$$

$$L = \frac{2 \times 10}{14} = 0.038$$

$$E = \frac{417.23}{0.038} = 10,979.74$$

$$T = E/18 = 10,979.74/18 = \text{about 610 seconds.}$$

Albrecht's Function Point Model

FPs - Measure the amount of functionality in a system as described by a specification

To compute UFC:

↳ def. some representations of the sys., the number of items of the foll. type

Ex: In/p's → items by user

Ext. O/p's - provided for the user

Ext. Inquiries - Interactive Blp's requiring a resp.

Ext. files - machine readable interfaces

Int. files - logical master files in the sys.

day. filename,
personal dict. name, ③.
additional file 1,

→ misspelled word report, # words processed,
errors ← for message ③.

→ word processed, errors so far ②.

→ da file, personal dict ②.
→ wait → ①.

$$UFC = \sum_{i=1}^{15} ((\text{No. of items of variety } i) \times (\text{weight}_i))$$

Weight factor

Item	Simple	Avg	Complex
E1	3	4	6
E2	4	5	7
E3	3	4	6
E4	7	10	15
E5	5	7	10

$$\begin{aligned}
 UFC &= (2 \times 4) + (1 \times 3) + (5 \times 2 + 7 \times 1) \\
 &\quad + (2 \times 4) + (2 \times 10) + (1 \times 7) \\
 &= 63
 \end{aligned}$$

$$TCF = 0.65 + 0.01 \sum_{i=1}^{14} f_i$$

Tech. Complexity factor

$$TCF = (0 \text{ to } 1.35)$$

$$FP = UFC \times TCF \quad (3)$$

F_1 to F_{14} .

Reliable backup to facilitate change.

$F_1 \rightarrow 0$ - incident

3 - average

5 - essential.

$$F_3, F_5, F_9, F_{11}, F_{12} \text{ & } F_{13} = 0$$

$$F_1, F_2, F_6, F_7, F_8, F_10, F_{14} = 3$$

$$F_4 \text{ & } F_10 = 5$$

$$TCF = 0.65 + 0.01(6+10+10) = 0.93$$

$$\text{As } UFC = 63$$

$$FP = 63 \times 0.93 = 59.$$

basis for an effort estimate.

It takes a developer an avg. of 29 persons ~~months~~ days of effort to impl. a FP.

Effort needed to complete spell checker

as $59 \text{ FPs} \times 2 \text{ person days} = 118 \text{ person days}$

Cocomo

1970, Barry Boehm, TRW, Comity
firms.

For effort estimation

barri Model: Little abt the project.

Integ. model: after reqs. are specified

Adv. Model: design is complete

Adv. Model: $E = a S^b F$

Barri form:

$$E = a S^b F$$

effort in person months

size measured in

Thousands of delivered KDSI

F → adjustment factor

($\gamma = 1$ in barri model)

Effort parameters

Model

Organic

a

b

2.4

1.05

Semi-detached

3.0

1.12

Embedded

3.6

1.20

→ for a telephone switch system
 KDS I = 5000
 S/w is embedded
 Initial Effort is

$$E = \frac{3.6(5000)}{1.2}$$

= 1,00,000 person months
 of effort

Duration:

$$D = a E^b$$

↑ Duration is months.

Our parameters

a	b
---	---

Mode

O	2.5	0.38
E	2.5	0.35
S	2.5	0.32

for organic project

$$D = 2.5(3000)^{0.38} = 52 \text{ months}$$

∴ Project requires 58 staffing
monthly for 52 months.

Cyclomatic Complexity

- Measures logical complexity of the program code
- Counts number of decisions in the given program code
- measure no. of linearly independent paths thru the (Pgm code) flow graph F .

Calculation:

- Using CF representation of the program code
- Nodes represent parts of the code with no branches
- Edges represent control flow transfers during program execution

For a program with flowgraph F ,

Method 1 For a program with flowgraph F ,

$$\text{Cyclomatic Complexity } CC = E - N + 2$$

$E \rightarrow \text{Edges}$ $N \rightarrow \text{Nodes}$

Method 2: No. of closed regions in the CFA + 1

M3 $v(P)/CC = P + 1$

$P = \text{Total no. of }$ predicate nodes
 \downarrow
 $\rightarrow 2 \text{ branches}$ conditional nodes

IF A = 25

THEN IF B > C

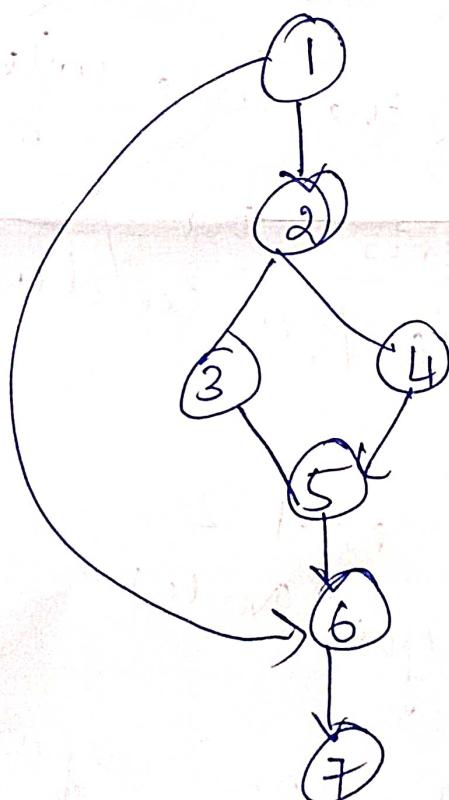
THEN A = B

ELSE A = C

END IF

END IF

PRINT A



$$\underline{M1}: CC = 2 + 1 \\ = 3$$

$$M2: 8 - 7 + 2$$

$$= 3$$

$$M3: P + 1 = 2 + 1 \\ = 3$$

McCabe suggests,

1 - 10 - well written module,
high testability with
less cost

10 - 20 - complex code

20 - 40 - very complex

740 - slightly complex

- Helps to focus more on the uncovered paths

- provides assurance to the developers
that all the paths have been tested
at least once.

- easy to apply & and measures the minimum
complexity & best areas of concentration
for testing

- always 7/1

- Max no. of independent paths through
the program code.

How to reduce CC:

- Write small functions

- Avoid flag parameters - need for decision
structures

Demerits of CC:

- From a measurement theory perspective, it is extremely doubtful that any of the assumptions corresponds to intuitive relationship about complexity
 - So $V(G)$ can't be used as a general complexity measure.
 - not at all clear that it paints a complete or accurate picture of program complexity.

Software Quality Metrics

If you can't measure
it, you can't manage it

Tom DeMarco, 1982

Software Crisis

- According to American Programmer, 31.1% of computer software projects get canceled before they are completed,
- 52.7% will overrun their initial cost estimates by 189%.
- 94% of project start-ups are restarts of previously failed projects.

Solution?

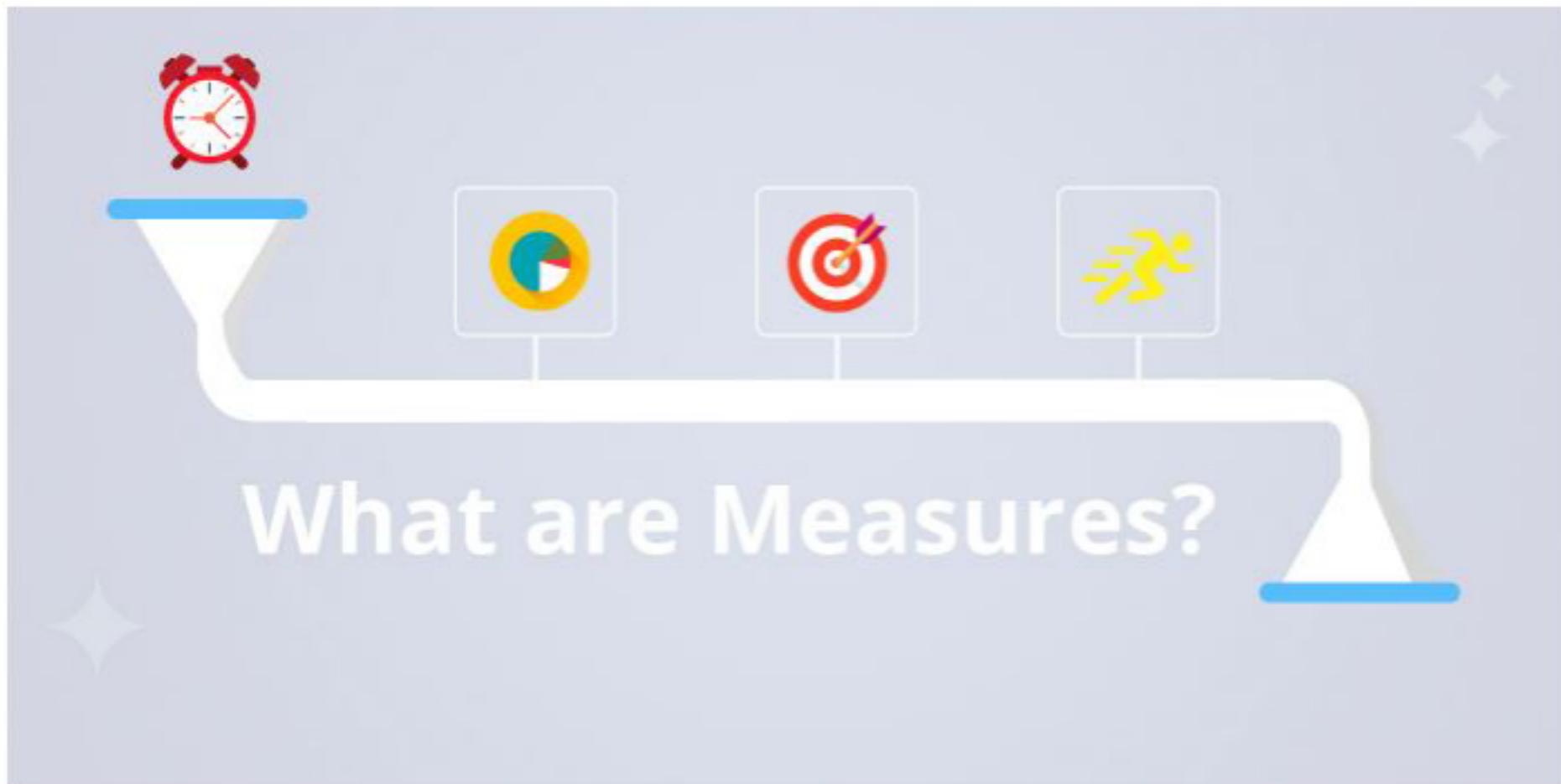
systematic approach to software development and measurement

Why Measure Software?

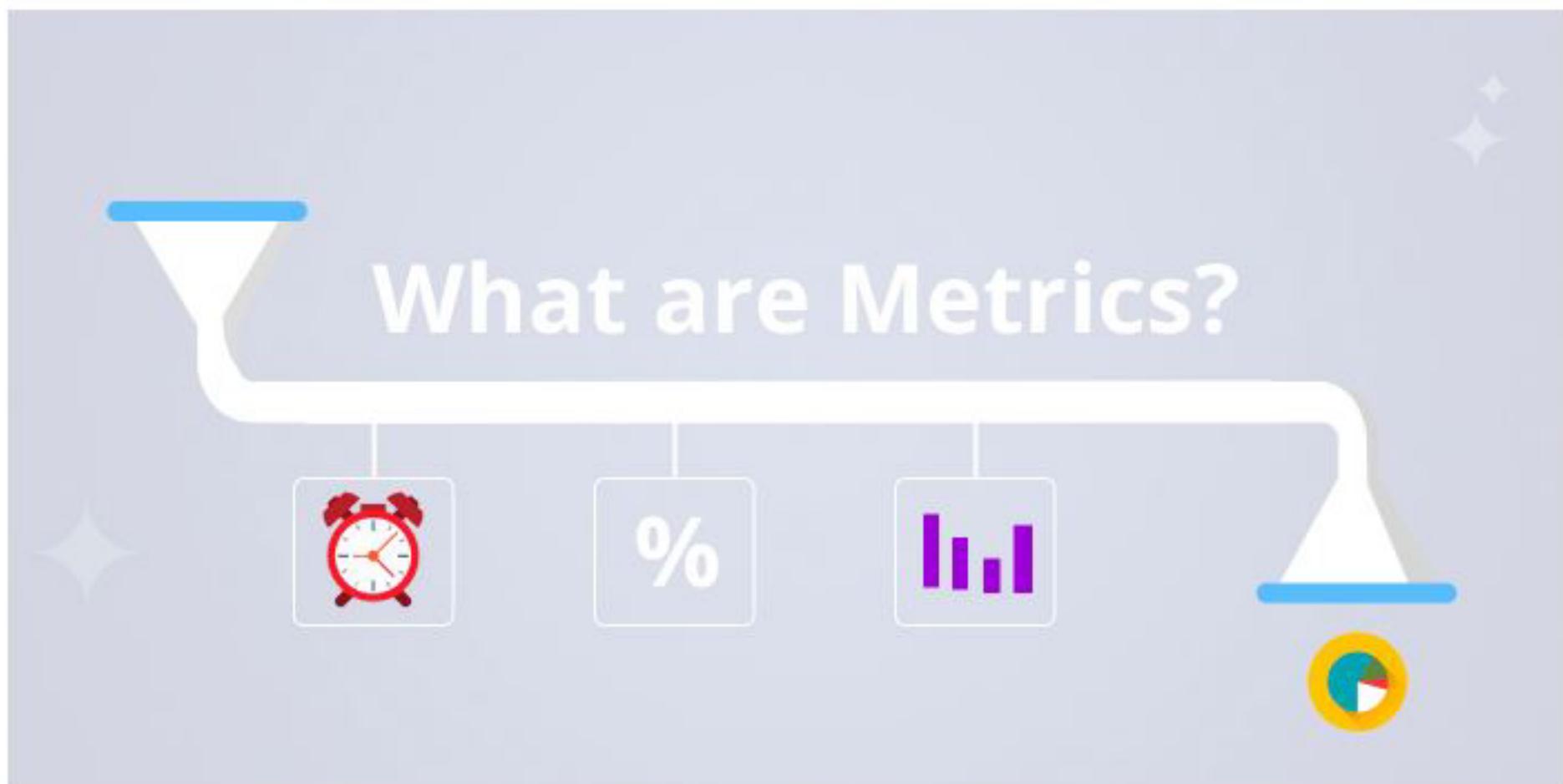
- To estimate development time and budget
- To improve **software quality**
 - If a software module *shares characteristics* of modules that are known often to fail, then these should be the focus of quality improvement

Measurement, Measures, Metrics

- Measurement
 - is the act of obtaining a measure
- Measure
 - provides a quantitative indication of the extent, amount, dimension, capacity, or size of some product or process attribute, E.g., Number of errors in the code
- Metric (A measure that is quantifiable)
 - is a quantitative measure of the degree to which a system, component, or process possesses a given attribute (*IEEE Software Engineering Standards 1993*) : *Software Quality* –
 - Relates the individual measures in some way.
 - E.g., temperature values over time, errors per KLOC, Number of errors found per person hours expended
- Reference:
<http://www.stsc.hill.af.mil/crosstalk/1995/03/Measure.asp>



<https://www.simplekpi.com/Blog/metrics-and-measures-a-definitive-guide>



<https://www.simplekpi.com/Blog/metrics-and-measures-a-definitive-guide>

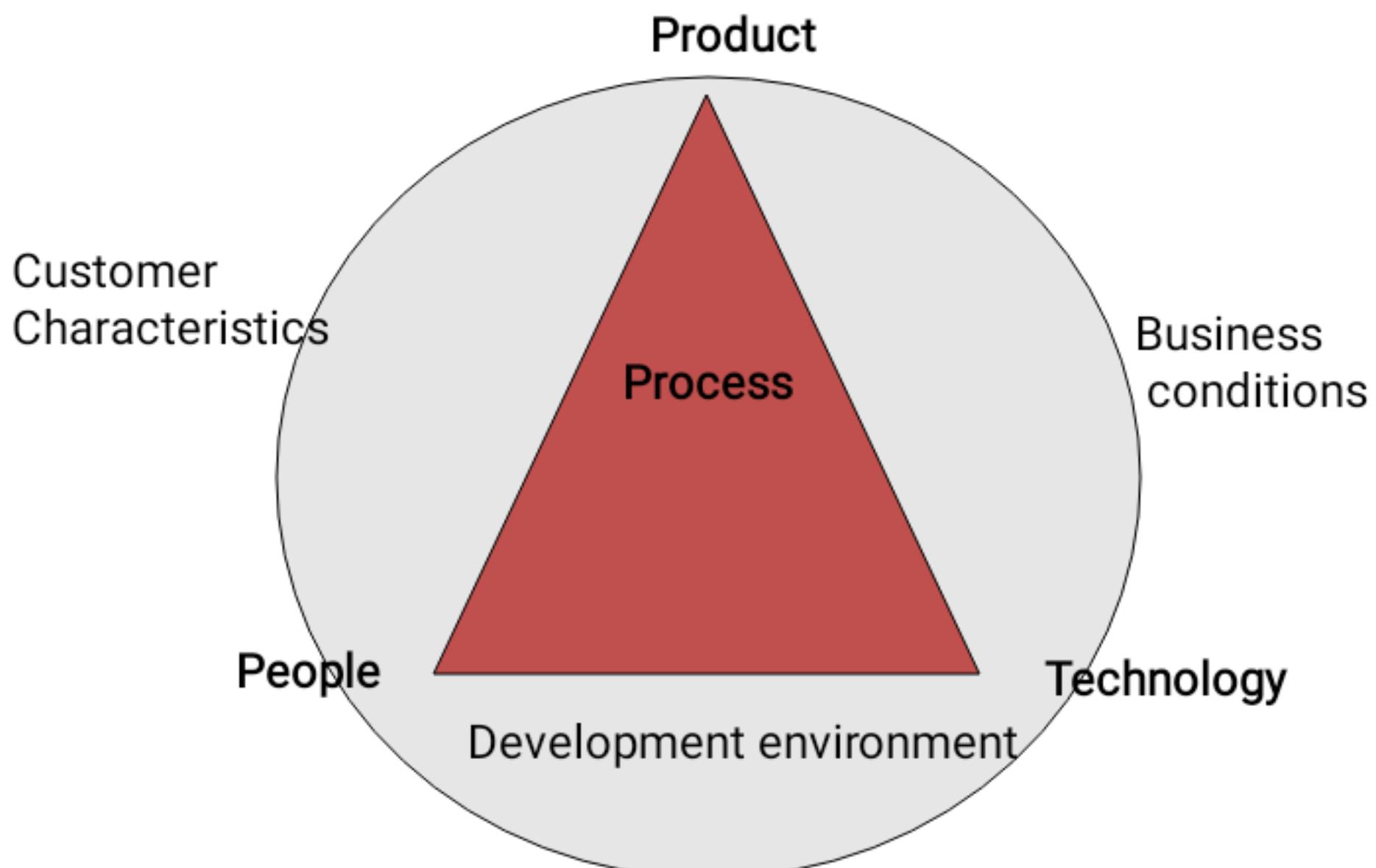
Metrics

Express in Numbers

Measurement provides a mechanism for
objective evaluation

Determinants for software quality and organizational effectiveness

Process connects 3 important factors that influence software quality and organizational performance



What to measure

- **Process**

Measure the efficacy of processes. What works, what doesn't.

- **Project**

Assess the status of projects. Track risk.
Identify problem areas. Adjust work flow.

- **Product**

Measure predefined product attributes

What to measure

- Process

Measure the efficacy of processes. What works, what doesn't.

- Code quality

- Programmer productivity

- Software engineer productivity

- Requirements,
 - design,
 - testing
 - and all other tasks done by software engineers

- Software

- Maintainability
 - Usability
 - And all other quality factors

- Management

- Cost estimation
 - Schedule estimation, Duration, time
 - Staffing

Process Metrics

- Process metrics are measures of the software development process, such as
 - Overall development time
 - Type of methodology used
- Process metrics are collected across all projects and over long periods of time.
- Their intent is to provide indicators that lead to long-term software process improvement.

Project Metrics

- Project Metrics are the measures of Software Project and are used to monitor and control the project. Project metrics usually show how project manager is able to estimate **schedule and cost**
- They enable a software project manager to:
 - Minimize the **development time** by making the adjustments necessary to avoid delays and potential problems and risks.
 - Assess **product cost** on an ongoing basis & modify the technical approach to improve cost estimation.

Project Metrics

- Example
 - Metrics collected from past projects help to establish time and effort estimates for current software projects
 - Production rates (in terms of models created, review hours, function points, delivered source lines)
 - Errors uncovered during each software engg. Task
- Software projects are assessed against the project metrics for improvement

Product metrics

- Product metrics are measures of the software product at any stage of its development, from requirements to installed system. Product metrics may measure:
 - How easy is the software to use
 - How easy is the user to maintain
 - The quality of software documentation
 - And more ..

Software Measurement Categories

- Direct Measures

- Of *software process* are
 - Cost and effort
- Of *product* are
 - Line of Code (LOC)
 - Execution Speed
 - Defects over specified period of time

- Indirect Measures

- Of *product* are
 - Functionality
 - Quality
 - Complexity
 - Efficiency
 - Reliability
 - Maintainability
 - Other Abilities

Software Metrics

Software cost and effort estimation
will never be an exact science

Software Metrics

- No estimation technique will be entirely accurate.
- Estimates fed back into the project, alter the behavior being modeled

Size Oriented Metrics

- Derived by normalizing quality and/or productivity measures by considering the *size* of software that has been produced.
- Source Lines of Code (SLOC) used as a normalizing factor for other measures
- Widely used in software industry but their validity and applicability is debated

Source Lines of Code (SLOC)

- Measures the number of physical lines of active code
- In general the higher the SLOC in a module the less understandable and maintainable the module is

Example Set of Size-oriented metrics

- Errors per KLOC (thousand lines of code)
- Defects per KLOC
- \$ per LOC
- Pages of Documentation per KLOC
- Errors per person-month
- LOC per person-month
- \$ per Page of documentation

Size-oriented productivity and quality metrics:

- Productivity = KLOC/person-month
- Quality = error/KLOC
- Cost = \$/KLOC
- Documentation = Pages doc./KLOC

Pros and cons

- Though widely used but not universally accepted as best way to measure a software process
- Controversy lies in using LOC as a key measure
- proponents claim
 - LOC easy to count
 - Many existing estimation models use LOC
 - Large literature & data based on LOC exists
- But opponents argue that
 - LOC measures are prog. language dependent
 - When considering productivity, LOC criteria penalizes well designed short programs
 - Can not accommodate non procedural languages
 - Planner must estimate LOC long before analysis and design

Examples of Metrics Usage

- Measure estimation skills of project managers (Schedule/ Budget)
 - Measure software engineers requirements/ analysis/design skills
 - Measure Programmers work quality
 - Measure testing quality
- And much more ...

IEEE definitions of software quality metrics

- (1) **A quantitative measure** of the degree to which an item possesses a given quality attribute.
- (2) **A function** whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which the software possesses a given quality attribute.

Main objectives of software quality metrics

1. **Facilitate** management control, planning and managerial intervention.

Based on:

- Deviations of actual from planned performance.
- Deviations of actual timetable and budget performance from planned.

2. **Identify** situations for development or maintenance process improvement (preventive or corrective actions). Based on:

- Accumulation of metrics information regarding the performance of teams, units, etc.

Software size (volume) measures

- **KLOC** – classic metric that measures the size of software by thousands of code lines.
- **Number of function points (NFP)** – a measure of the development resources (human resources) required to develop a program, based on the functionality specified for the software system.

Process metrics categories

- Software process quality metrics
 - Error density metrics
 - Error severity metrics
- Software process timetable metrics
- Software process error removal effectiveness metrics
- Software process productivity metrics

Is KLOC enough ?

- What about number of errors (error density)?
- What about types of errors (error severity) ?
- A mixture of KLOC, density, and severity is an ideal quality metric to programmers quality of work and performance

An example

- 2 different project programmers are working on two similar modules
- Programmer A- produced 342 errors during software process before release
- Programmer B- Produced 184 errors
- Which Programmer do you think is better ?

An example

- It really depends on the types of errors found (severity) and not only the number of errors (Density)
- One error of high severity might be more important than hundreds of other types of errors

Error density metrics

Code	Name	Calculation formula
CED	Code Error Density	$CED = \frac{NCE}{KLOC}$
DED	Development Error Density	$DED = \frac{NDE}{KLOC}$

NCE = The number of code errors detected by code inspections and testing.

NDE = Total number of development (design and code) errors detected in the development process.

Error severity metrics

Code	Name	Calculation formula
ASCE	Average Severity of Code Errors	$ASCE = \frac{WCE}{NCE}$
ASDE	Average Severity of Development Errors	$ASDE = \frac{WDE}{NDE}$

NCE = The number of code errors detected by code inspections and testing.

NDE = total number of development (design and code) errors detected in the development process.

WCE = weighted total code errors detected by code inspections and testing.

WDE = total weighted development (design and code) errors detected in development process.

Software process timetable metrics

Code	Name	Calculation formula
TTO	Time Table Observance	$TTO = \frac{MSOT}{MS}$
ADMC	Average Delay of Milestone Completion	$ADMC = \frac{TCDAM}{MS}$

MSOT = Milestones completed on time.

MS = Total number of milestones.

TCDAM = Total Completion Delays (days, weeks, etc.) for all milestones.

Time Table Metric Example

- TTO
 - Milestones are Requirements, Analysis, Design, Implementation, and Testing
 - Milestones completed in time are Requirements and analysis only
 - $TTO = 2/5$

Time Table Metric Example

- ADMC
 - Requirements (One week delay, Analysis (Three weeks delay, Design (Two weeks delay, Implementation (Six weeks delay), and Testing (Two weeks delay)
 - Total Delay is 14 Weeks
 - ADMS = 14/5

Error removal effectiveness metrics

Code	Name	Calculation formula
DERE	Development Errors Removal Effectiveness	$\text{DERE} = \frac{\text{NDE}}{\text{NDE} + \text{NYF}}$

NDE = total number of development (design and code) errors detected in the development process.

NYF = number software failures detected during a year of maintenance service.

Defect Removal Efficiency

- It is a measure of the filtering ability of the quality assurance and control activities as they are applied through out the process framework.
- $DRE = E / (E + D)$
 - E = # of errors found before delivery
 - D = # of defects found after delivery
- Ideal value of DRE is 1
- Realistically D will be greater than 0
- As E increases, DRE begins to approach 1

Error removal efficiency

- DRE

- Number of errors detected at design and coding stages is 100
- Number of errors detected after one year of maintenance service is 500
- DERE= $100/(100+500)$

Redefining DRE

- DRE can also be applied on each process framework activity and hence find the team's ability to assess errors before they are passed to next activity or software engineering task.
- $DRE = E_i / (E_i + E_{i+1})$
 - E_i = errors in activity i
 - E_{i+1} = errors in activity $i+1$ that were not discovered in activity i

Process productivity metrics

Code	Name	Calculation formula
DevP	Development Productivity	$DevP = \frac{DevH}{KLOC}$
FDevP	Function point Development Productivity	$FDevP = \frac{DevH}{NFP}$
CRe	Code Reuse	$Cre = \frac{ReKLOC}{KLOC}$
DocRe	Documentation Reuse	$DocRe = \frac{ReDoc}{NDoc}$

DevH = Total working hours invested in the development of the software system.

ReKLOC = Number of thousands of reused lines of code.

ReDoc = Number of reused pages of documentation.

NDoc = Number of pages of documentation.

White box and Black box Testing

Slides adapted from various web sources with grateful acknowledgement of the many others who made their course materials freely available online.

convention wisdom

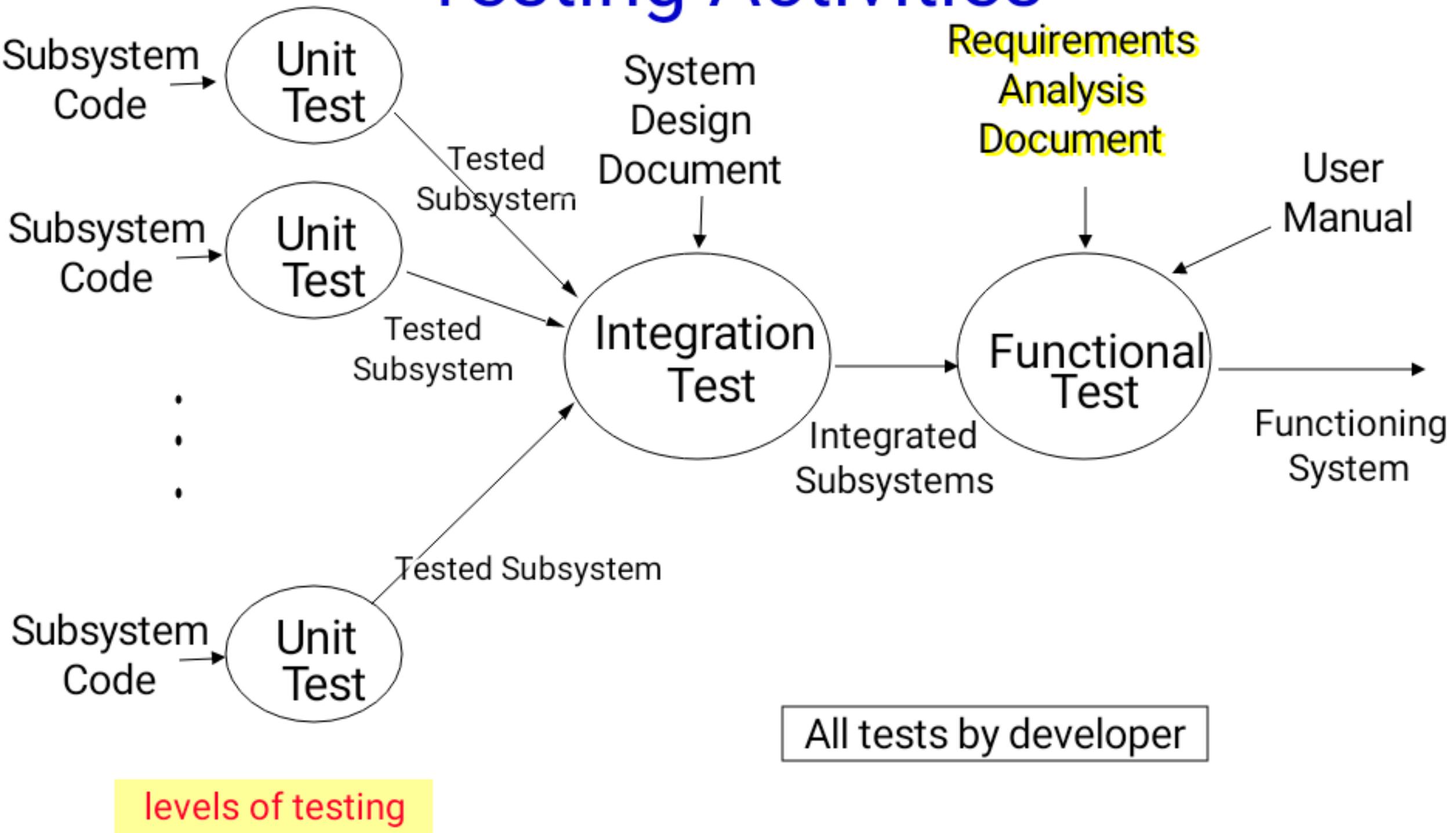
95% of errors are in 5% of the code

...or maybe 80% of the errors are in 20% of the code...

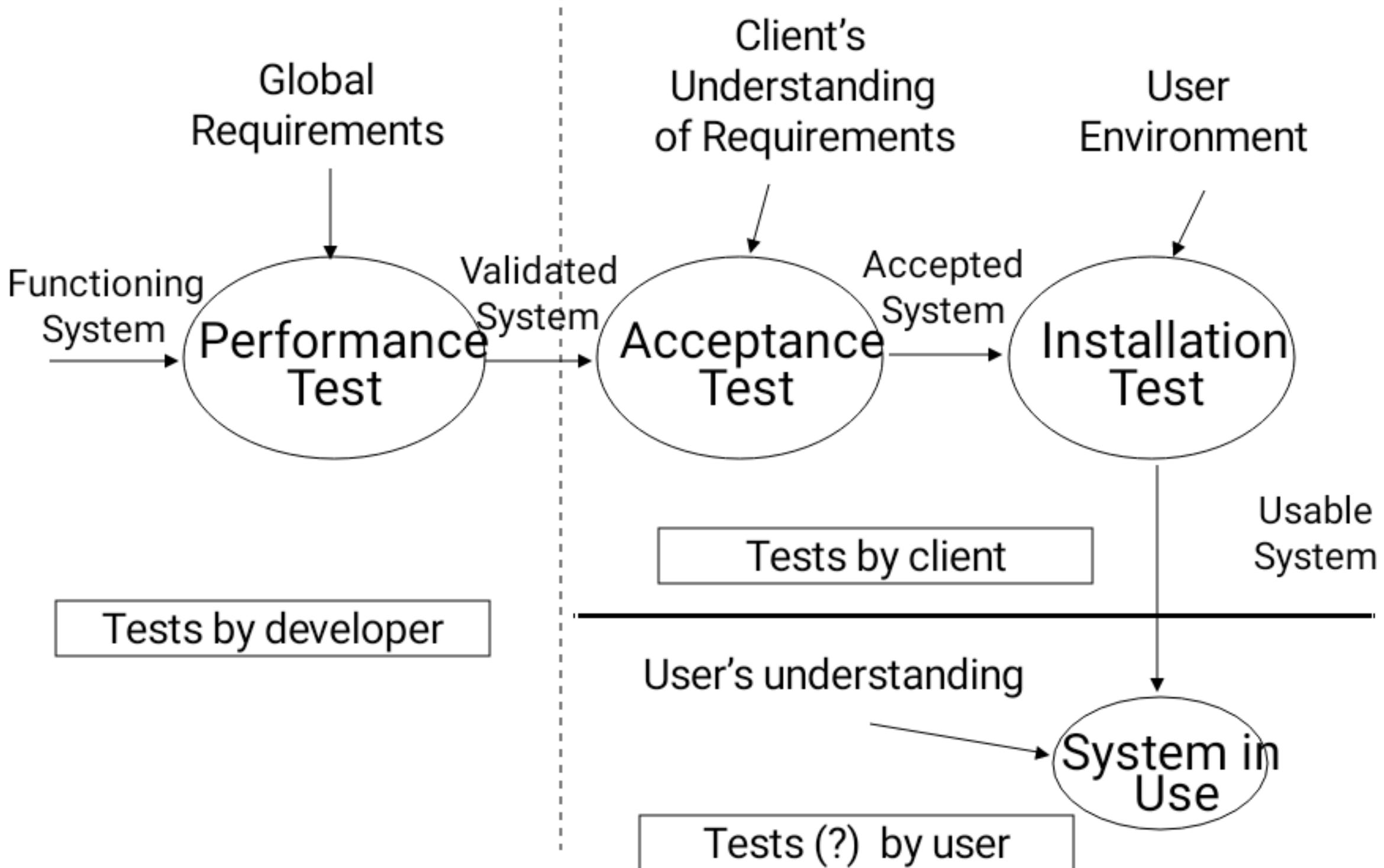
Testing: can I find that 5% or 20% via a test suite

Test suite is a container that has a set of tests which helps testers in executing and reporting the test execution status. It can take any of the three states namely Active, Inprogress and completed

Testing Activities



Testing Activities continued



Few terms...

Error

- A mistake made by a programmer during coding

Defect

- An error found during the unit testing in the development phase

Bug

- An error found during the testing phase

Failure

- When an error is found at an end user's end

Fault - the cause of failure - an unintended or incorrect behavior by an application program.

Basic Definitions – (Book)

- Software Error – made by programmer
 - Syntax (grammatical) error
 - Logic error (multiply instead of adding two operands)
- Software Fault –
 - All software errors may not cause software faults
 - That part of the software may not be executed
- Software Failures – Here's the interest.
 - A software fault becomes a software failure when/if it is activated.
 - Faults may be found in the software due to the way the software is executed or
 - Other constraints on the software's execution, such as execution options.

How does it work?

Error

Error – due to wrong logic, syntax, or loop

Types and examples:

Communication errors - no menu provided in the software, no help instructions, no save button

Missing command error – due to low typing speed, short deadlines..

Grammatical incorrect sentences and misspelled words..

Calculation errors - occur due to coding errors, bad logic, incorrect formulae, function call issues, data type mismatch

Bug

Example - When a user writes a report or article in a word processing software, and it crashes suddenly, the user will lose all the work if they don't press the save button before. This will have a negative impact on the productivity of the user.

- **Typos** are also bugs

Types:

- Logic bugs
- Algorithmic bugs
- Resource bugs

Reason Behind Bug:

- Missing Coding
- Wrong Coding
- Extra Coding

Failure

- A software failure can occur if a human puts a **wrong input value**.
- time, environmental conditions, including a strong magnetic field, pollution, electronic fields, radiation burst, etc., can cause failure in the firmware or hardware.
- a failure can also be caused **intentionally** in the system by an individual.

Fault

- If left untreated, may lead to failures in the working of the deployed code
- Can be prevented by adopting programming techniques, development methodologies, peer review, and code analysis.

Types of faults:

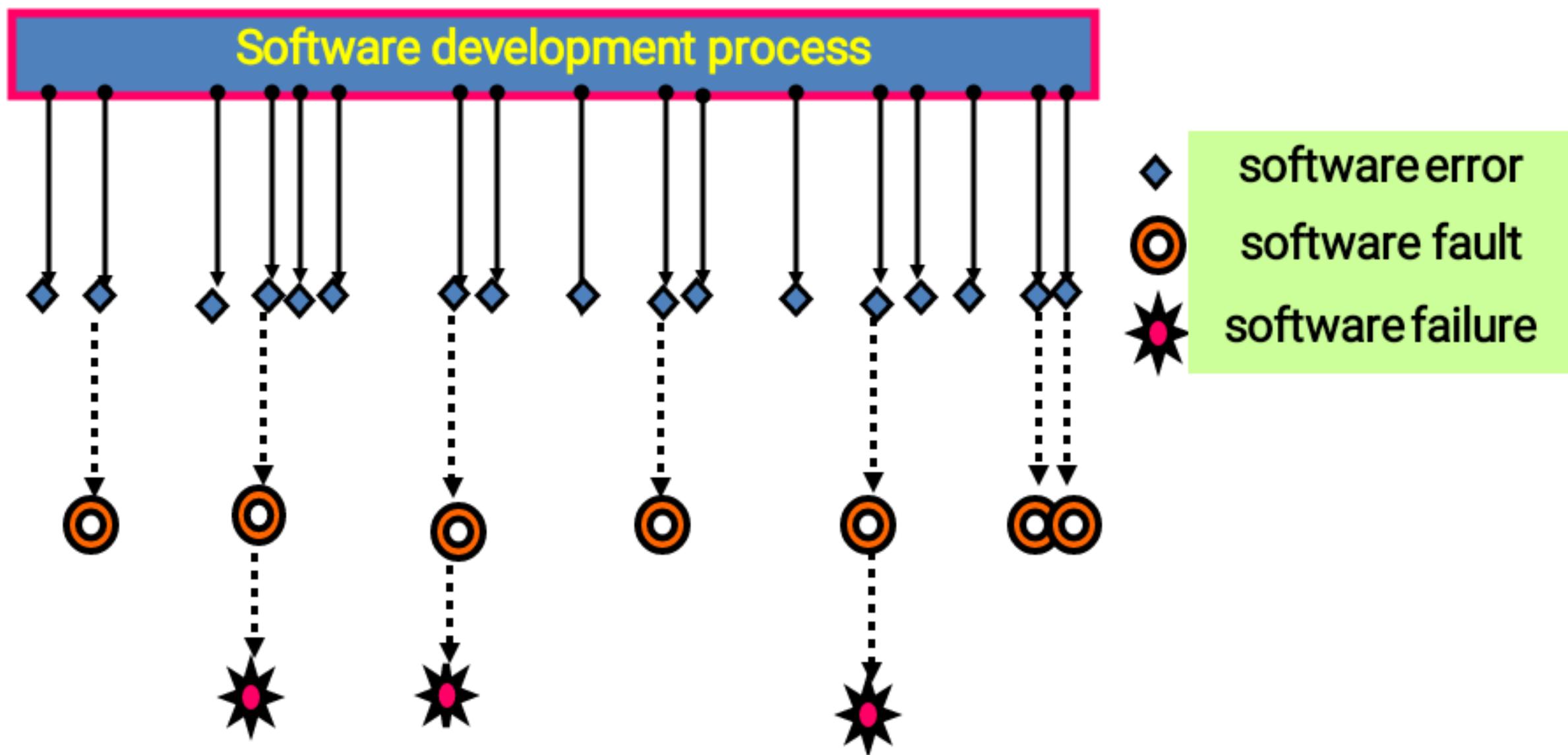
- Algorithm fault
- Syntax fault
- Computational fault - combining floating point and integer variables
- Omission fault - initialization of the variable is not done at the starting point.

Erroneous State (“Error”)

Algorithmic Fault

Mechanical Fault

Software errors, software faults and software failures



Defects

- deviation or variation of the software

Types:

- Arithmetic Defect
- Syntax Defects - escapes a symbol
- Logical Defects
- Interface Defects – e.g. complicated interfaces, platform-based interfaces

Types of tests

- **Black box:** test based on interface, through interface
- **White box:** test based on code, through code

Testing strategy can/should include almost all approaches!

Black Box = non developer, outside testing folks
in your case who?

White Box = developer, part of development process
in your case who?

Unit Testing (White Box)

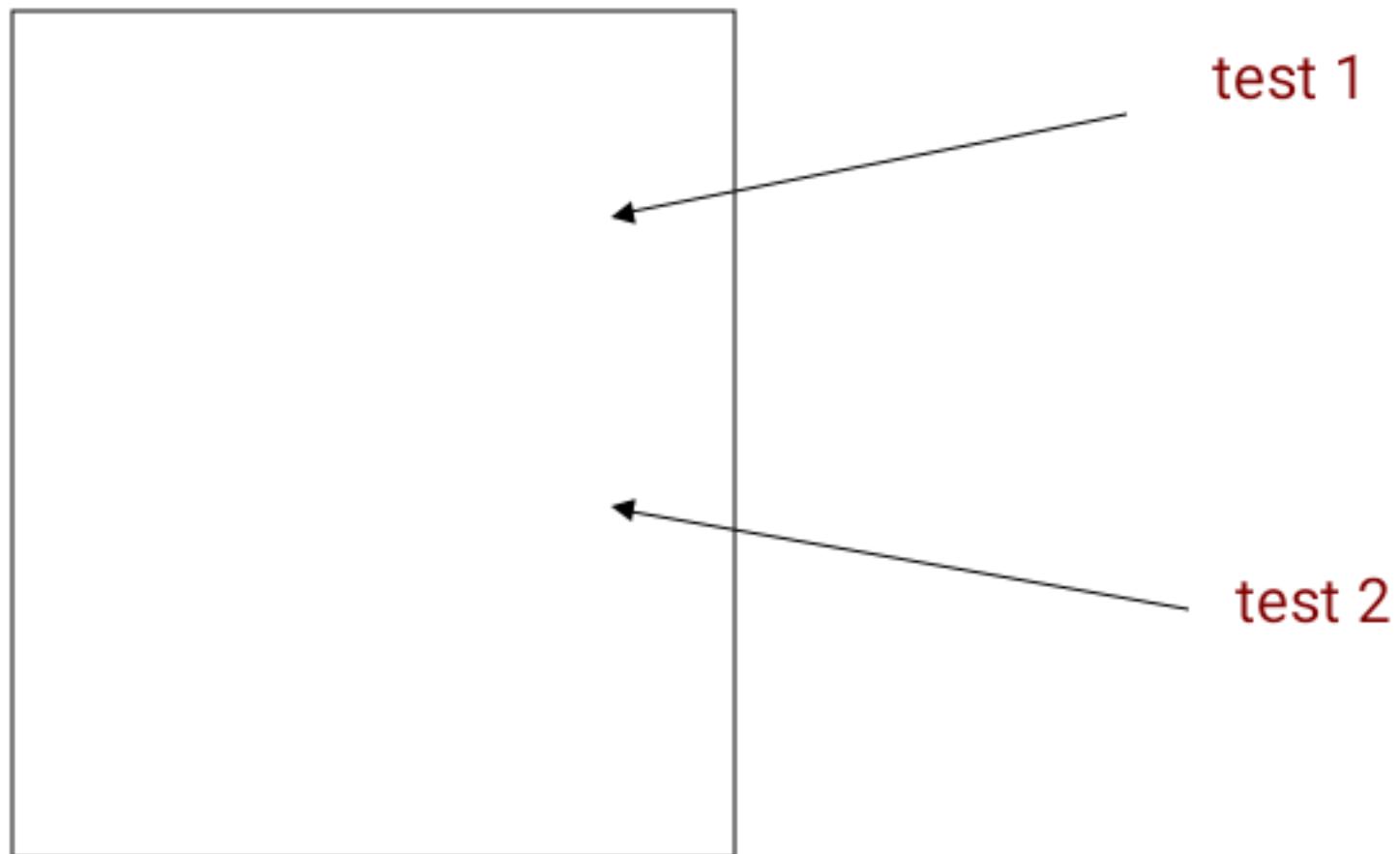
- White-box testing technique proposed by Tom McCabe
- Individual components are tested.
- It is a path test.
- To focus on a relatively small segment of code and aim to exercise a high percentage of the internal path

White-box Testing

- Uses the control structure part of component-level design to derive the test cases
- These test cases
 - Guarantee that all independent paths within a module have been exercised at least once
 - Exercise all logical decisions on their true and false sides
 - Execute all loops at their boundaries and within their operational bounds
 - Exercise internal data structures to ensure their validity
“Bugs lurk in corners and congregate at boundaries”

White box tests

Based on code



Example: ordered list of ints

```
class ordInts {  
public: ...  
  
private:  
    int vals[1000];  
    int maxElements=1000;  
  
    ...  
}
```

Example: ordered list of ints

```
bool testMax()
{
    L=create();
    num=maxElements;
    for (int i=0; i<=num; i++)
        print i
    L.insert(i)
        print maxElements;
}
```

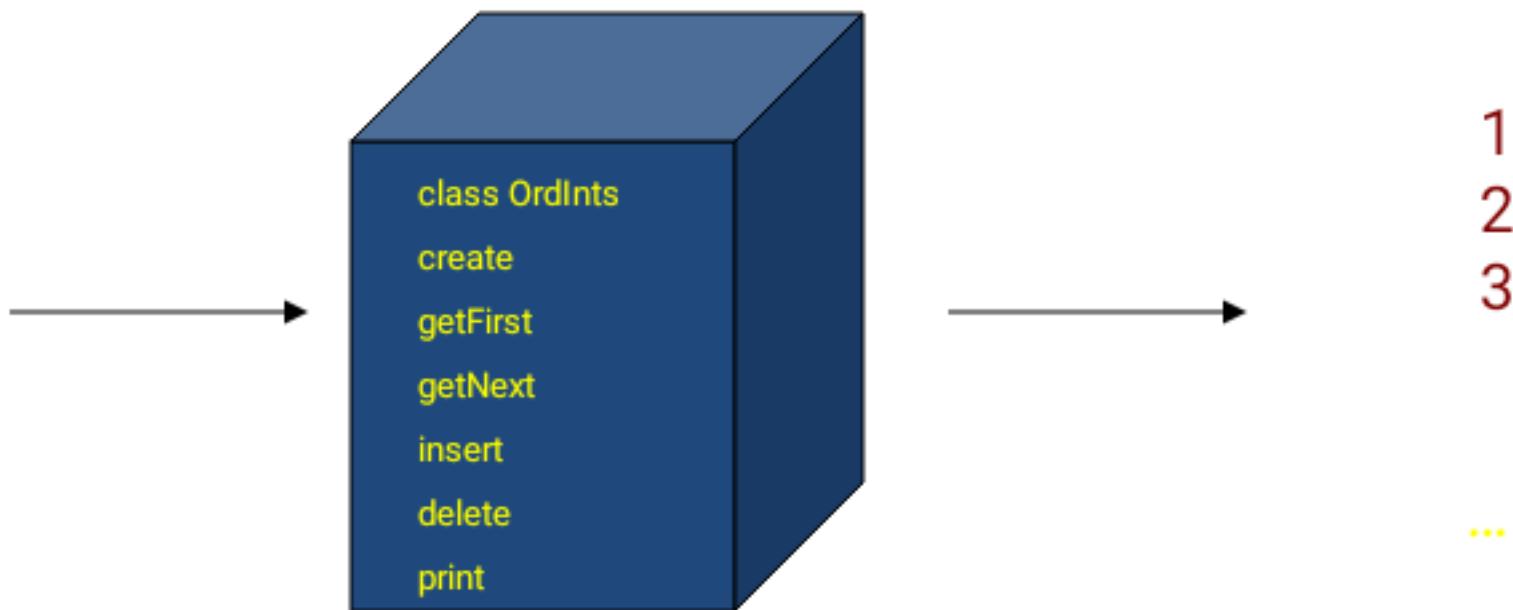
White box testing

```
1  "
2 You
3 Are
4 Not
5 Expected
6 to
7 Understand
8 This
9 "
10 How
11 26
12 Lines
13 of
14 Code
15 Changed
16 the
17 World
18 Edited
19 by
20 Torie
21 Bosch
22 with
23 illustrations
24 by
25 Kelly
26 Chudler
```

Example: ordered list of ints

```
L=create()  
L.insert(1)  
L.insert(2)  
L.insert(3)
```

...



```
L.insert(1001)  
p=L.getFirst()  
print(p)  
p=L.getNext(p)  
print p
```

...

Code coverage (White Box)

- Code coverage – individual modules
- Code coverage based on complexity – test of the risks, tricky part of code (e.g., Unix “**you are not expected to understand this**” code)

```
1 "
2 You
3 Are
4 Not
5 Expected
6 to
7 Understand
8 This
9 "
10      How
11      26
12      Lines
13      of
14      Code
15      Changed
16      the
17      World
18          Edited
19          by
20          Torie
21          Bosch
22          With
23          illustrations
24          by
25          Kelly
26          Chudler
```

Coverage Metrics (White Box)

- Coverage metrics
 - ***Statement coverage***: all statements in the programs should be executed at least once
 - ***Branch coverage***: all branches in the program should be executed at least once
 - ***Path coverage***: all execution paths in the program should be executed at least once
- The best case would be to execute all paths through the code, but there are some problems with this:
 - the number of paths increases fast with the number of branches in the program
 - the number of executions of a loop may depend on the input variables and hence may not be possible to determine
 - most of the paths can be infeasible

Statement Coverage

- Choose a test set T such that by executing program P for each test case in T , each basic statement of P is executed at least once
- Executing a statement once and observing that it behaves correctly is not a guarantee for correctness, but it is an heuristic
 - this goes for all testing efforts since in general checking correctness is undecidable

```
bool isEqual(int x, int y)
{
    if (x = y)
        z := false;
    else
        z := true;
    return z;
}
```

```
int max(int x, int y)
{
    if (x > y)
        return x;
    else
        return y;
}
```

Statement Coverage

```
areTheyPositive(int x, int y)
{
    if (x >= 0)
        print("x is positive");
    else
        print("x is negative");
    if (y >= 0)
        print("y is positive");
    else
        print("y is negative");
}
```

Following test set will give us statement coverage:

$$T_1 = \{(x=12,y=5), (x=-1,y=35),\\ (x=115,y=-13),(x=-91,y=-2)\}$$

Smaller set of test cases?

Statement Coverage

```
areTheyPositive(int x, int y)
{
    if (x >= 0)
        print("x is positive");
    else
        print("x is negative");
    if (y >= 0)
        print("y is positive");
    else
        print("y is negative");
}
```

Following test set will give us statement coverage:

$$T_1 = \{(x=12,y=5), (x=-1,y=35),\\ (x=115,y=-13),(x=-91,y=-2)\}$$

There are smaller test cases which will give us statement coverage too:

$$T_2 = \{(x=12,y=-5), (x=-1,y=35)\}$$

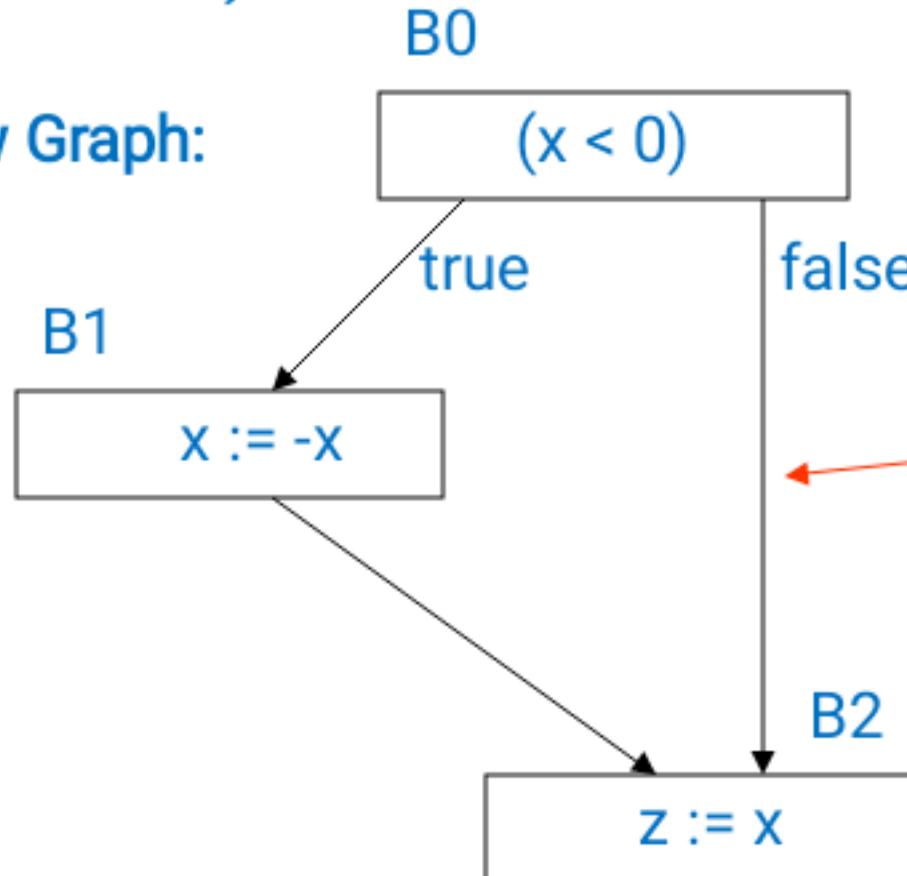
There is a difference between these two test sets though

Statement vs. Branch Coverage

```
assignAbsolute(int x)
{
    if (x < 0)
        x := -X;
    z := X;
}
```

Consider this program segment, the test set $T = \{x=1\}$ will give statement coverage, however not branch coverage

Control Flow Graph:



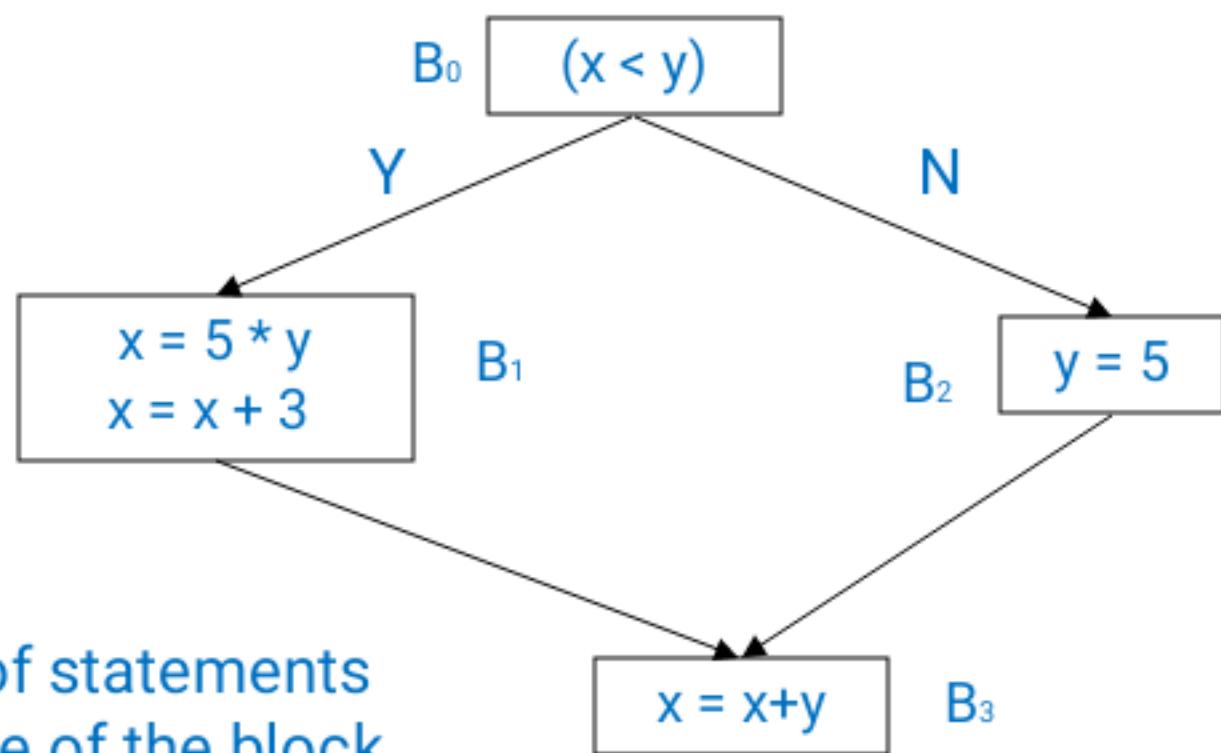
Test set $\{x=1\}$ does not execute this edge, hence, it does not give branch coverage

Test case for both statement and branch coverage?

Control Flow Graphs (CFGs)

- Nodes in the control flow graph are basic blocks
 - A **basic block** is a sequence of statements always entered at the beginning of the block and exited at the end
- Edges in the control flow graph represent the control flow

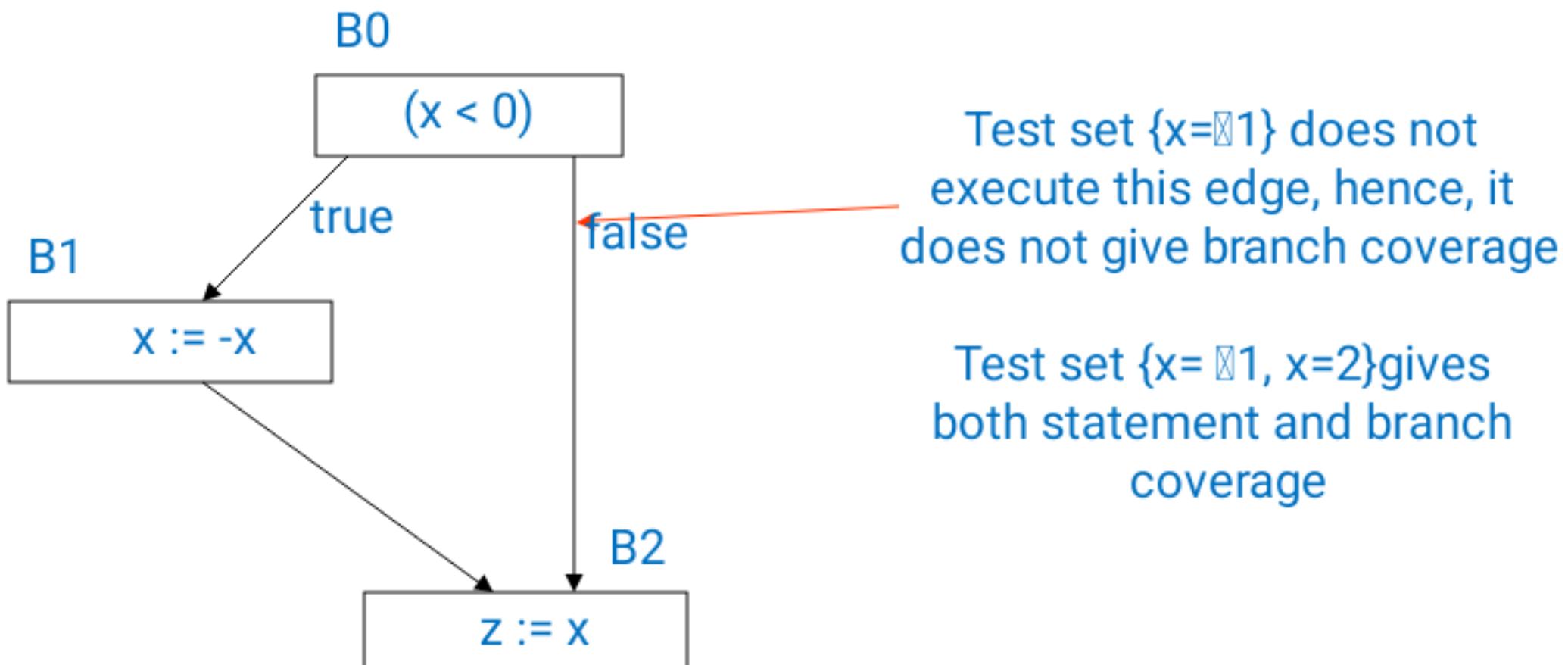
```
if (x < y) {  
    x = 5 * y;  
    x = x + 3;  
}  
else  
    y = 5;  
x = x+y;
```



- Each block has a sequence of statements
- No jump from or to the middle of the block
- Once a block starts executing, it will execute till the end

Branch Coverage

- Construct the control flow graph
- Select a test set T such that by executing program P for each test case d in T , each edge of P 's control flow graph is traversed at least once



Path coverage

- How many execution paths have been exercised by tests?
- 100% path coverage is usually impossible
- aim to cover **common** paths and error prone (**complex**) paths
- aim to break code with tests – **good testers are not liked by developers....**

Basis Path Testing

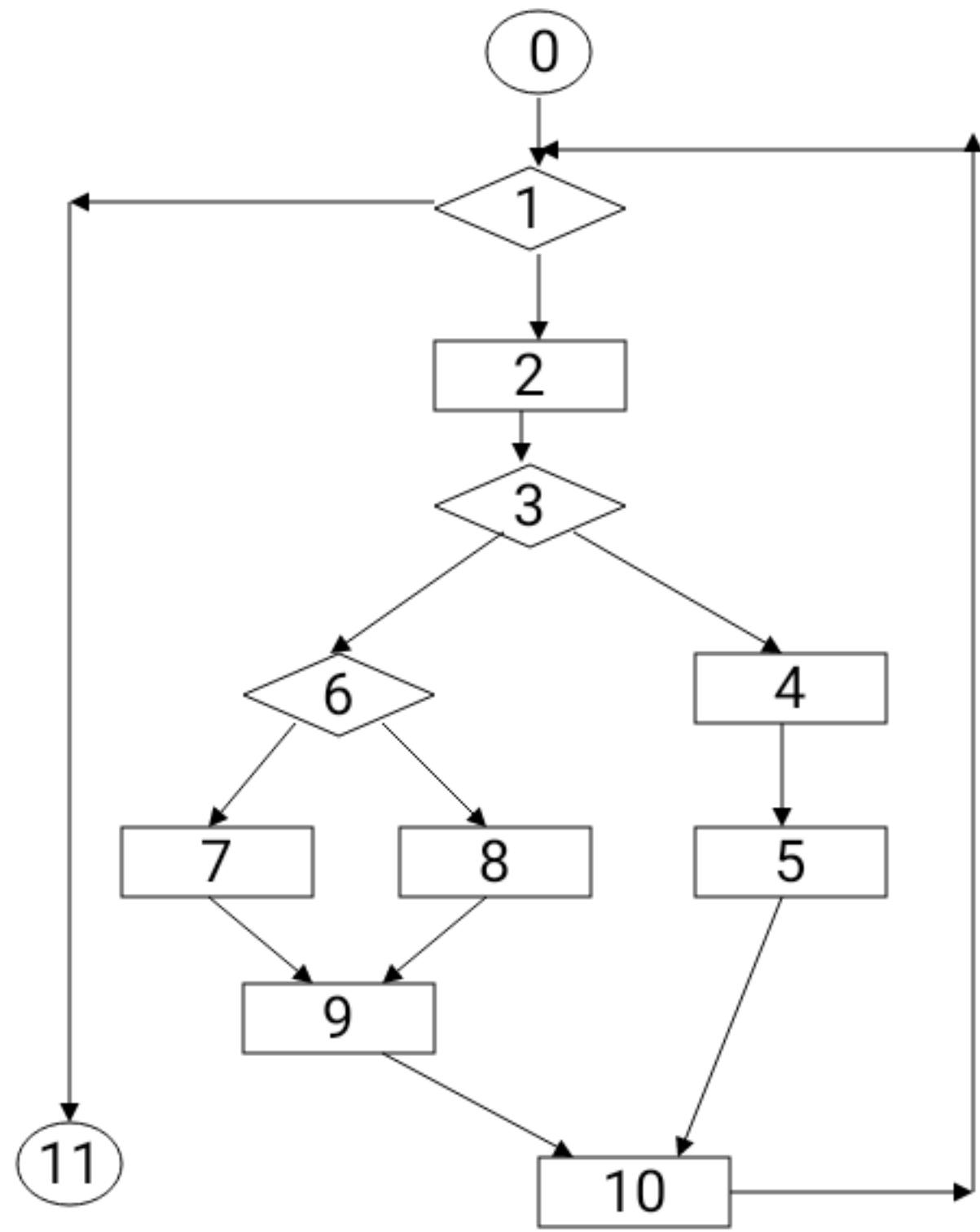
- Enables the test case designer to derive a **logical complexity** measure of a procedural design
- Uses this measure as a guide for defining a basis set of execution paths
- Test cases derived to exercise the **basis set** are guaranteed to execute every statement in the program at least one time during testing

Flow Graph Notation - revisit

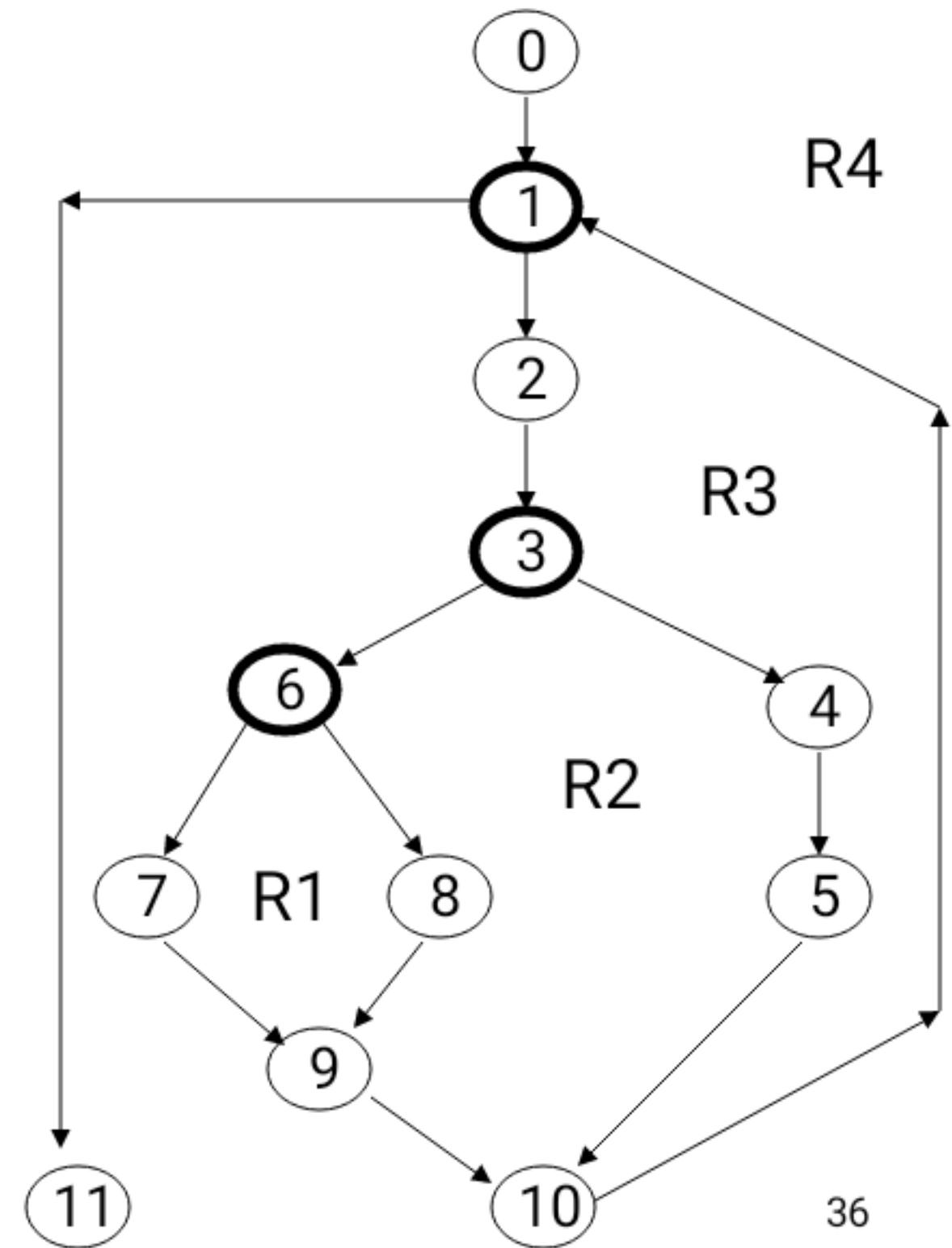
- A circle in a graph represents a node, which stands for a sequence of one or more procedural statements
- A node containing a simple conditional expression is referred to as a predicate node
 - Each compound condition in a conditional expression containing one or more Boolean operators (e.g., and, or) is represented by a separate predicate node
 - A predicate node has two edges leading out from it (True and False)
- An edge, or a link, is a an arrow representing flow of control in a specific direction
 - An edge must start and terminate at a node
 - An edge does not intersect or cross over another edge
- Areas bounded by a set of edges and nodes are called regions
- When counting regions, include the area outside the graph as a region, too

Flow Graph Example -1

FLOW CHART



FLOW GRAPH



Independent Program Paths

- Defined as a path through the program from the start node until the end node that introduces at least one new set of processing statements or a new condition (i.e., new nodes)
- Must move along at least one edge that has not been traversed before by a previous path
- **Basis set** for flow graph on previous slide
 - Path 1: 0-1-11
 - Path 2: 0-1-2-3-4-5-10-1-11
 - Path 3: 0-1-2-3-6-8-9-10-1-11
 - Path 4: 0-1-2-3-6-7-9-10-1-11
- The number of paths in the basis set is determined by the cyclomatic complexity

Cyclomatic Complexity

- Provides a quantitative measure of the logical complexity of a program
- Defines the number of independent paths in the basis set
- Provides an upper bound for the number of tests that must be conducted to ensure all statements have been executed at least once
- Can be computed three ways
 - The number of regions
 - $V(G) = E - N + 2$, where E is the number of edges and N is the number of nodes in graph G
 - $V(G) = P + 1$, where P is the number of predicate nodes in the flow graph G
- Results in the following equations for the example flow graph
 - Number of regions = 4
 - $V(G) = 14 \text{ edges} - 12 \text{ nodes} + 2 = 4$
 - $V(G) = 3 \text{ predicate nodes} + 1 = 4$

Deriving the Basis Set and Test Cases

- Using the design or code as a foundation, draw a corresponding flow graph
- Determine the cyclomatic complexity of the resultant flow graph
- Determine a basis set of linearly independent paths
- Prepare test cases that will force execution of each path in the basis set

Path Coverage

- Select a test set T such that by executing program P for each test case d in T , all paths leading from the initial to the final node of P 's control flow graph are traversed

Path Coverage

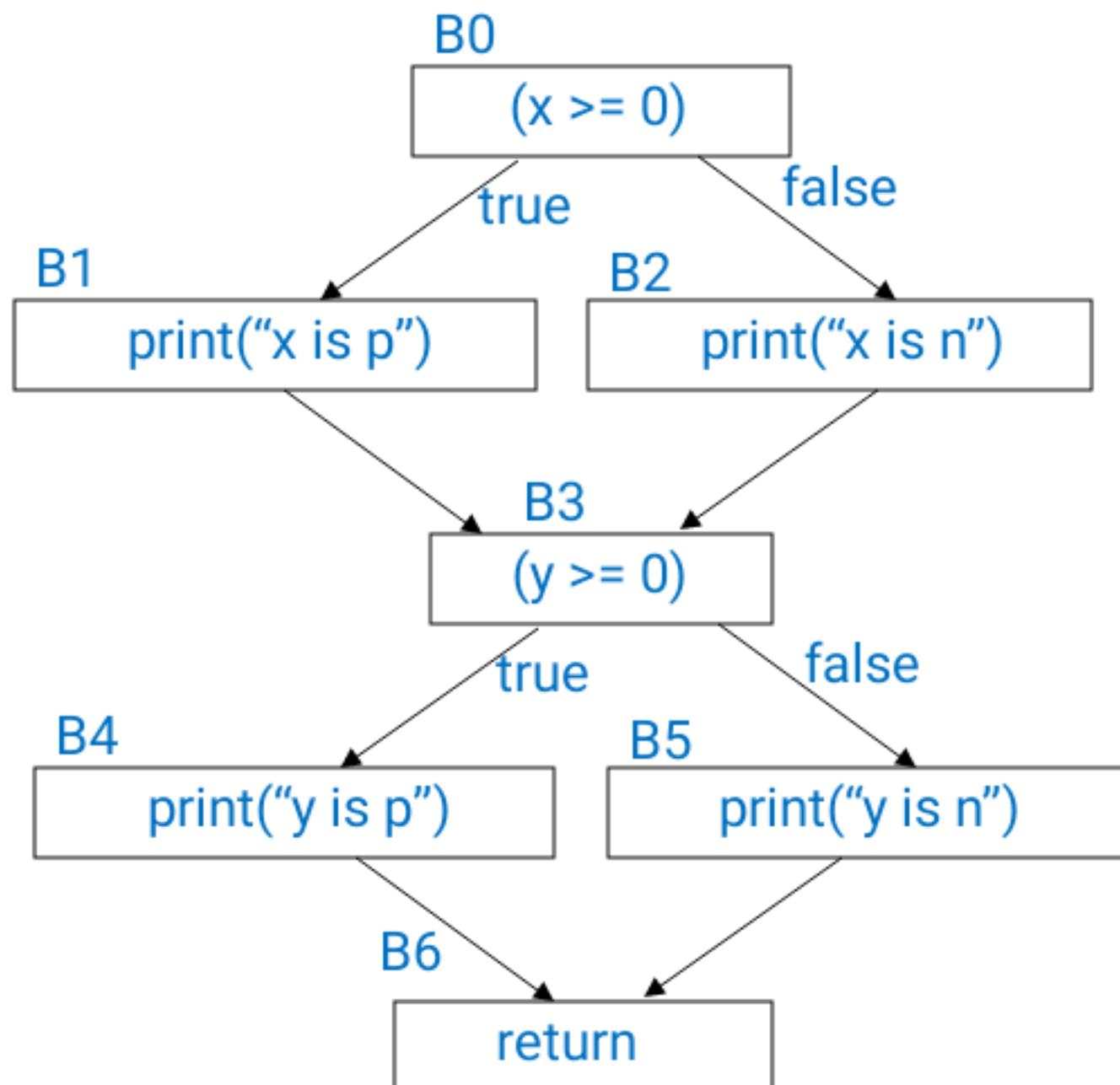
```
areTheyPositive(int x, int y)
{
    if (x >= 0)
        print("x is positive");
    else
        print("x is negative");
    if (y >= 0)
        print("y is positive");
    else
        print("y is negative");
}
```

Test set:

$T_2 = \{(x=12, y=-5), (x=-1, y=35)\}$
gives both branch and statement
coverage but it does not give path
coverage

Set of all execution paths: $\{(B0, B1, B3, B4, B6), (B0, B1, B3, B5, B6), (B0, B2, B3, B4, B6), (B0, B2, B3, B5, B6)\}$

Test set T_2 executes only paths: $(B0, B1, B3, B5, B6)$ and $(B0, B2, B3, B4, B6)$



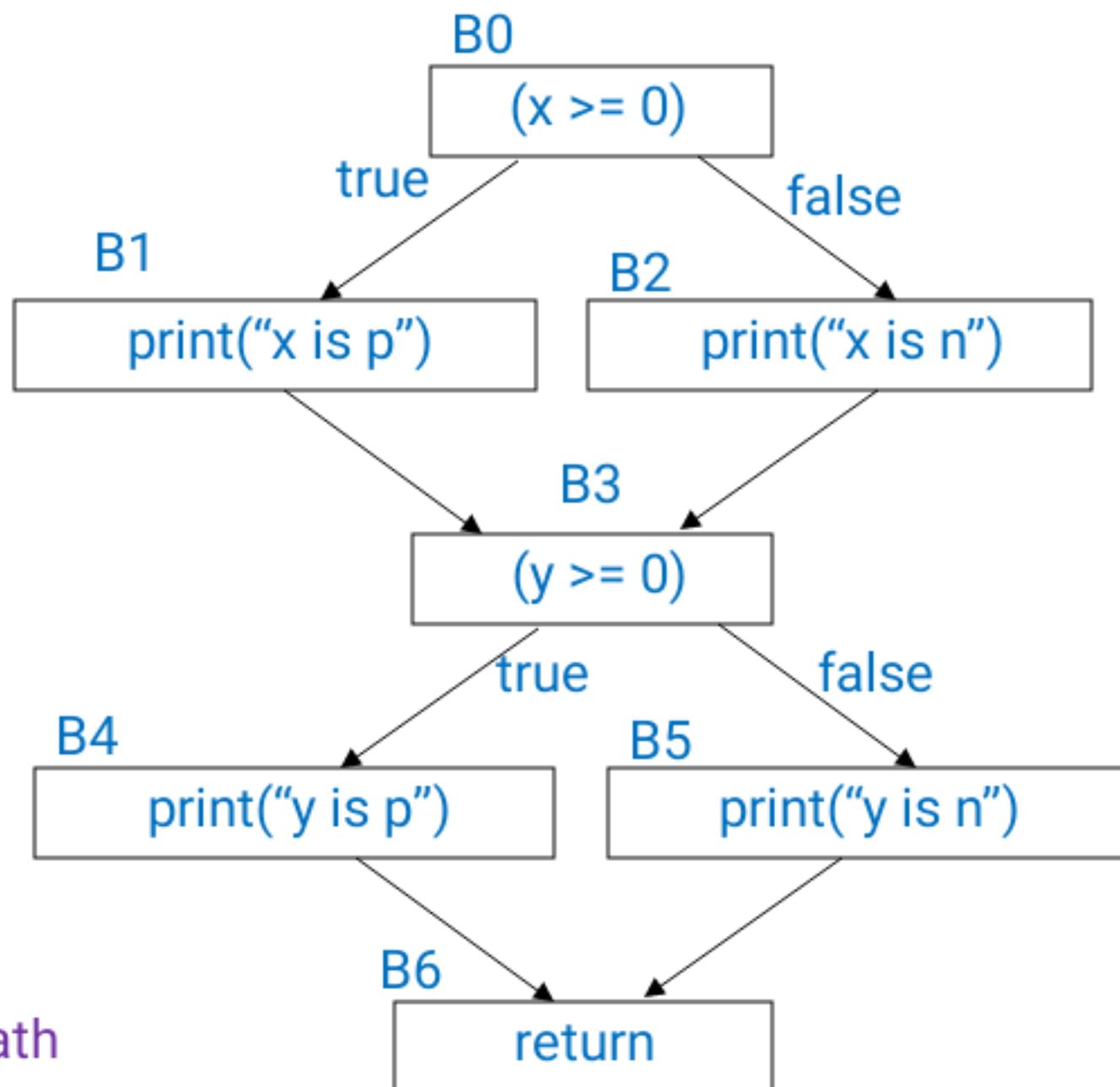
Path Coverage

```
areTheyPositive(int x, int y)
{
    if (x >= 0)
        print("x is positive");
    else
        print("x is negative");
    if (y >= 0)
        print("y is positive");
    else
        print("y is negative");
}
```

Test set:

$$T_1 = \{(x=12, y=5), (x=-1, y=35),\\ (x=115, y=-13), (x=-91, y=-2)\}$$

gives both branch, statement and path coverage



Path Coverage

- Number of paths is exponential in the number of conditional branches
 - testing cost may be expensive
- Note that every path in the control flow graphs may not be executable
 - It is possible that there are paths which will never be executed due to dependencies between branch conditions
- In the presence of cycles in the control flow graph (for example loops) we need to clarify what we mean by path coverage
 - Given a cycle in the control flow graph we can go over the cycle arbitrary number of times, which will create an infinite set of paths
 - Redefine path coverage as: each cycle must be executed 0, 1, ..., k times where k is a constant (k could be 1 or 2)

Path coverage

- an attempt to use test input that will pass once over each path in the code
 - What would be path testing for `daysInMonth(month, year)`?
 - some ideas:
 - error input: `year < 1`, `month < 1`, `month > 12`
 - one month from `[1, 3, 5, 7, 10, 12]`
 - one month from `[4, 6, 8, 9, 11]`
 - month 2
 - in a leap year, not in a leap year

Path coverage examples

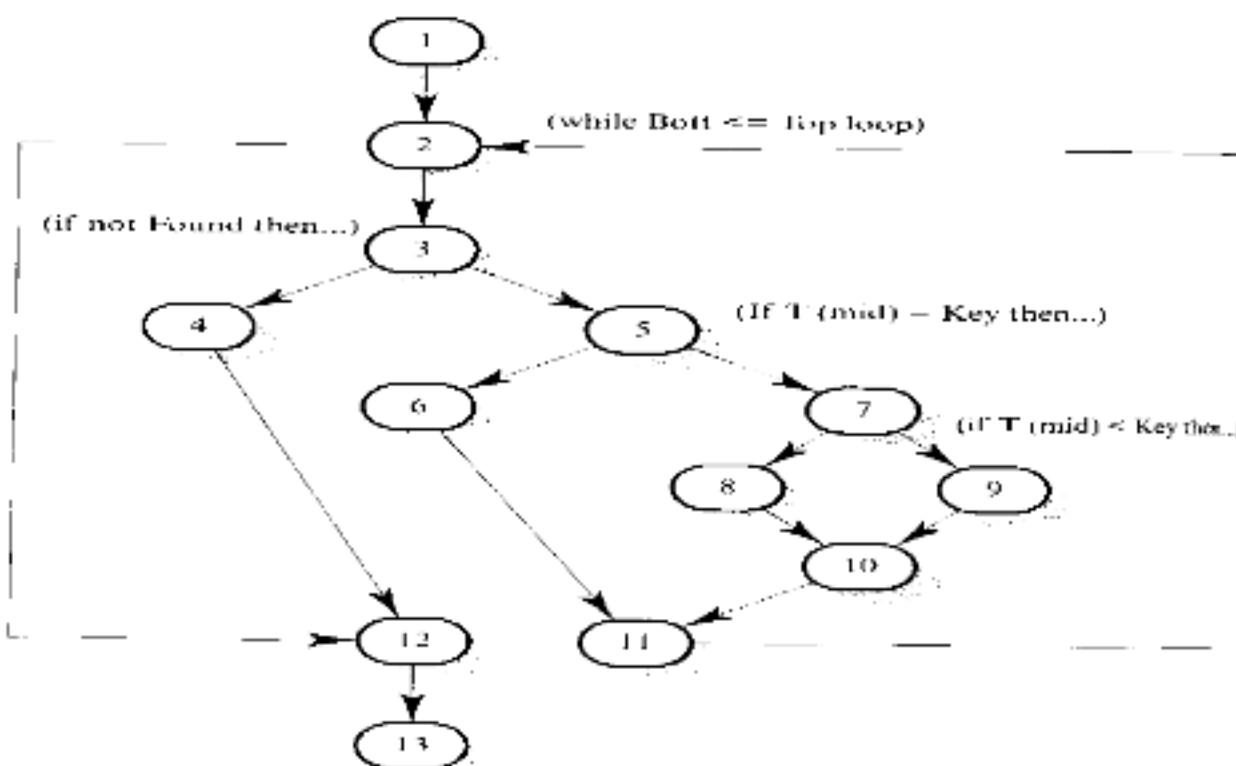


Figure 23.15
Flow graph for a binary search routine.

Path coverage examples

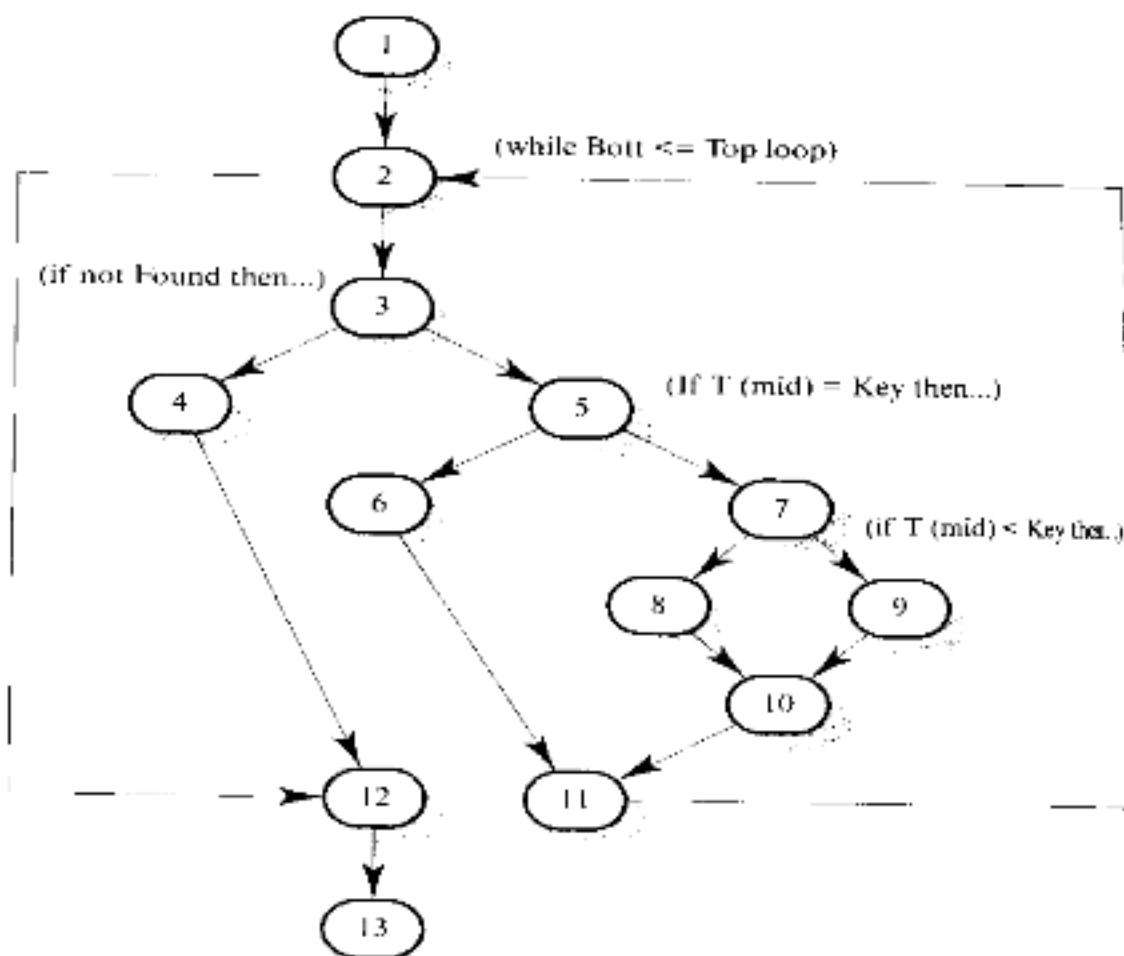


Figure 23.15
Flow graph for a binary search routine.

Binary Search Paths

Basis Set:

- 1, 2, 3, 4, 12, 13
- 1, 2, 3, 5, 6, 11, 2, 12, 13
- 1, 2, 3, 5, 7, 8, 10, 11, 2, 12, 13
- 1, 2, 3, 5, 7, 9, 10, 11, 2, 12, 13
- Test cases should be derived so that all of these paths are executed
- A dynamic program analyser may be used to check that paths have been executed

Condition Coverage (White Box)

- In the branch coverage we make sure that we execute every branch at least once
 - For conditional branches, this means that, we execute the TRUE branch at least once and the FALSE branch at least once
- Conditions for conditional branches can be compound boolean expressions
 - A compound boolean expression consists of a combination of boolean terms combined with logical connectives AND, OR, and NOT
- Condition coverage:
 - Select a test set T such that by executing program P for each test case d in T , (1) each edge of P 's control flow graph is traversed at least once **and** (2) each boolean term that appears in a branch condition takes the value TRUE at least once and the value FALSE at least once
- Condition coverage is a refinement of branch coverage

Software Testing Strategies

- A strategic approach to testing
- Test strategies for conventional software
- Test strategies for object-oriented software
- Validation testing
- System testing
- The art of debugging

To tell
somebody that
they are wrong
is called
criticism.

To do so
officially is
called testing.

Introduction

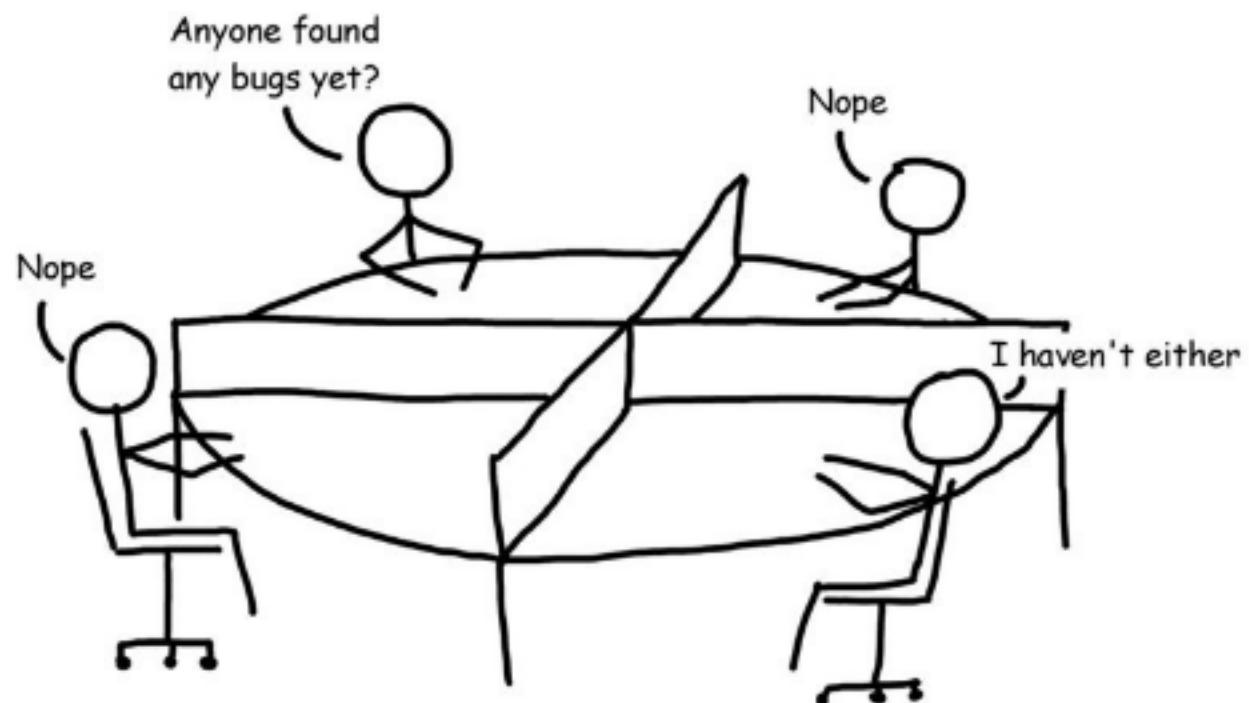
- A strategy for software testing integrates the design of software test cases into a well-planned series of steps that result in successful development of the software
- The strategy provides a **road map** that describes the steps to be taken, when, and how much effort, time, and resources will be required
- The strategy incorporates test planning, test case design, test execution, and test result collection and evaluation
- The strategy provides **guidance** for the practitioner and a set of milestones for the manager
- Because of time pressures, progress must be **measurable** and problems must surface as early as possible

Test Case

- The test case is defined as **a group of conditions** under which a tester determines whether a software application is working as per the customer's requirements or not.
- A test case is a defined format for software testing required to check if a particular application/software is working or not.
- Test case designing includes **preconditions, case name, input conditions, and expected result**. A test case is a first level action and derived from test scenarios.

A Strategic Approach to Testing

A test team before automation tools were invented



General Characteristics of Strategic Testing

- To perform effective testing, a software team should conduct effective **formal technical reviews**
- Testing begins at the **component level** and work outward toward the integration of the entire computer-based system
- Different testing techniques are appropriate at different points in time
- Testing is conducted by the developer of the software and (for large projects) by an independent test group
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy

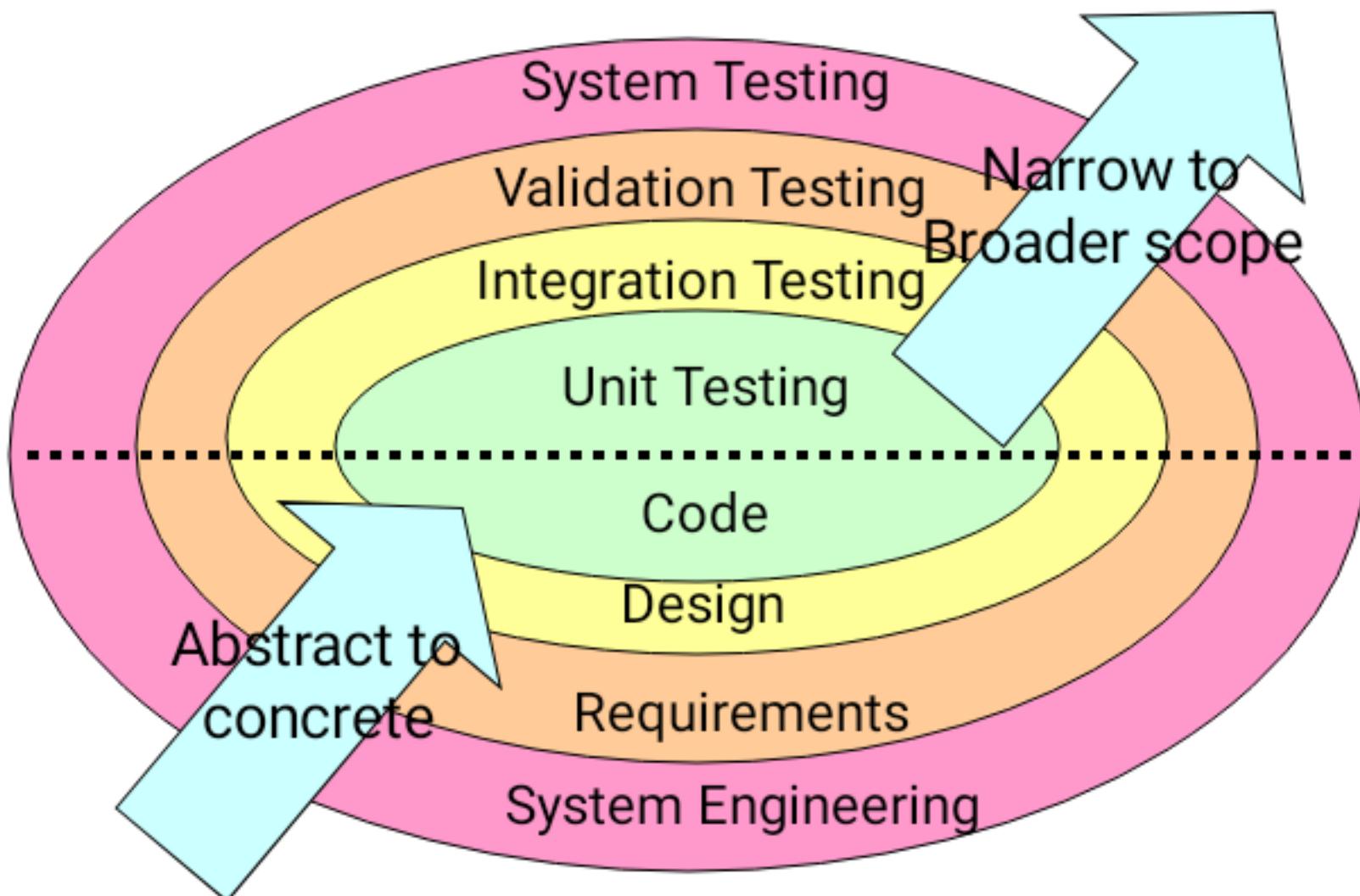
Verification and Validation

- Software testing is part of a broader group of activities called verification and validation that are involved in software quality assurance
- Verification (Are the algorithms coded correctly?)
 - The set of activities that ensure that software correctly implements a specific function or algorithm
- Validation (Does it meet user requirements?)
 - The set of activities that ensure that the software that has been built is traceable to customer requirements

Organizing for Software Testing

- Testing should aim at "breaking" the software
- Common misconceptions
 - The developer of software should do no testing at all
 - The software should be given to a secret team of testers who will test it unmercifully
 - The testers get involved with the project only when the testing steps are about to begin
- Reality: Independent test group
 - Removes the inherent problems associated with letting the builder test the software that has been built
 - Removes the conflict of interest that may otherwise be present
 - Works closely with the software developer during analysis and design to ensure that thorough testing

A Strategy for Testing Conventional Software



TESTER

DEVELOPER

techacade.me

Levels of Testing for Conventional Software

- Unit testing
 - Concentrates on each component/function of the software as implemented in the source code
- Integration testing
 - Focuses on the design and construction of the software architecture
- Validation testing
 - Requirements are validated against the constructed software
- System testing
 - The software and other system elements are tested as a whole

Testing Strategy applied to Conventional Software

- **Unit testing**
 - Exercises specific paths in a component's control structure to ensure **complete coverage and maximum error detection**
 - Components are then assembled and integrated
- **Integration testing**
 - Focuses on inputs and outputs, and how well the components fit together and work together
- **Validation testing**
 - Provides final assurance that the software meets all functional, behavioral, and performance requirements
- **System testing**
 - Verifies that all system elements (software, hardware, people, databases) mesh properly and that overall system function and performance is achieved

Testing Strategy applied to Object-Oriented Software

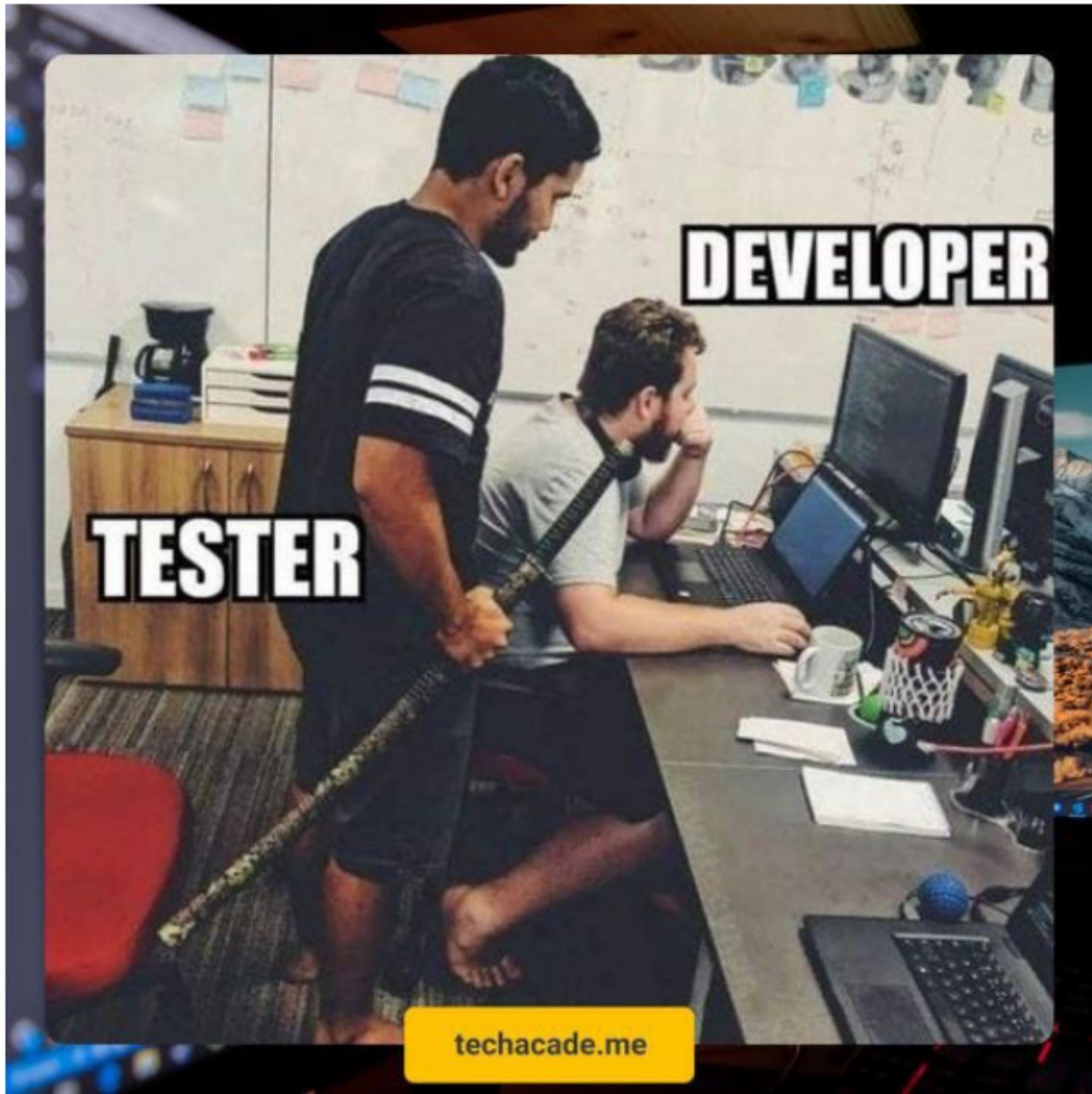
- Must broaden testing to include detections of errors in analysis and design models
- Unit testing loses some of its meaning and integration testing changes significantly
- Use the same philosophy but different approach as in conventional software testing
- Test "in the small" and then work out to testing "in the large"
 - Testing in the small involves **class attributes and operations**; the main focus is on **communication and collaboration** within the class
 - Testing in the large involves a **series of regression tests** to uncover errors due to communication and collaboration among classes
- Finally, the system as a whole is tested to detect errors in fulfilling requirements

When is Testing Complete?

- There is **no definitive** answer to this question
- Every time a user executes the software, the program is being tested
- Sadly, testing usually stops when a project is running out of time, money, or both
- One approach is to divide the test results into various **severity levels**
 - Then consider testing to be complete when certain levels of errors no longer occur or have been repaired or eliminated

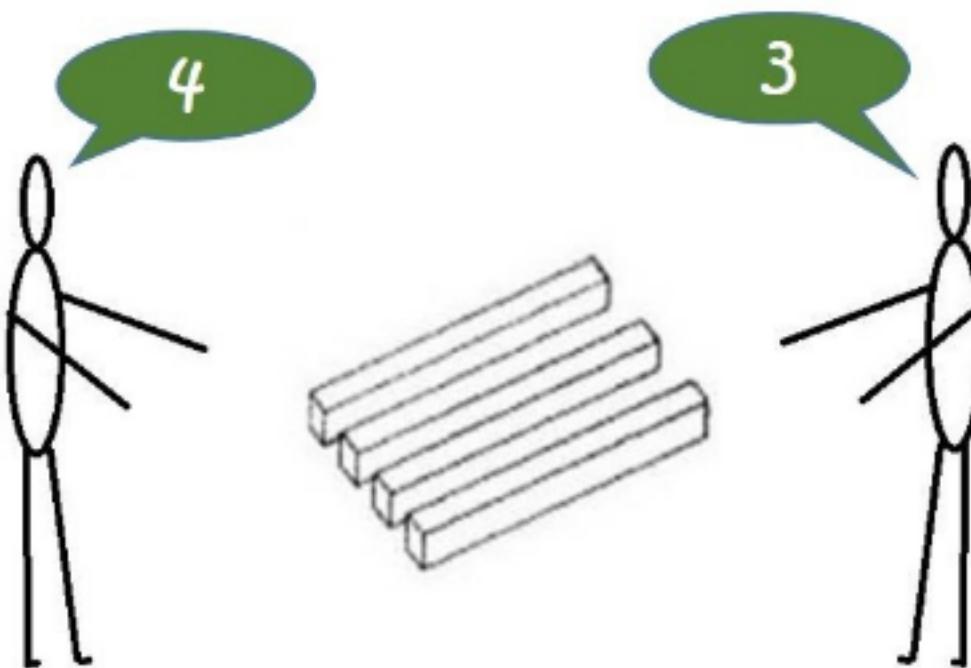
Ensuring a Successful Software Test Strategy

- Specify product requirements in a quantifiable manner long before testing commences
- State **testing objectives** explicitly in measurable terms
- Understand the user of the software (through use cases) and develop a profile for each user category
- Develop a testing plan that emphasizes rapid cycle testing to get quick feedback to control quality levels and adjust the test strategy
- Build **robust software** that is designed to **test itself** and can diagnose certain kinds of errors
- Use effective formal technical reviews as a filter prior to testing to reduce the amount of testing required
- Conduct **formal technical reviews** to assess the test strategy and test cases themselves
- Develop a **continuous improvement** approach for the testing process through the gathering of metrics



Developer

Tester



TestingWhiz
Code Less, Test More

Test Strategies for Conventional Software

Unit Testing

- Focuses testing on the **function** or software module
- Concentrates on the **internal processing logic** and data structures
- Is simplified when a module is designed with high cohesion
 - Reduces the number of test cases
 - Allows errors to be more easily predicted and uncovered
- Concentrates on critical modules and those with **high cyclomatic complexity** when testing resources are limited

Targets for Unit Test Cases

- Module interface
 - Ensure that information flows properly into and out of the module
- Local data structures
 - Ensure that data stored temporarily maintains its integrity during all steps in an algorithm execution
- Boundary conditions
 - Ensure that the module operates properly at boundary values established to limit or restrict processing
- Independent paths (basis paths)
 - Paths are exercised to ensure that all statements in a module have been executed **at least once**
- Error handling paths
 - Ensure that the algorithms respond correctly to specific error conditions

Common Computational Errors in Execution Paths

- Misunderstood or incorrect arithmetic precedence
- Mixed mode operations (e.g., int, float, char)
- Incorrect initialization of values
- Precision inaccuracy and round-off errors
- Incorrect symbolic representation of an expression (int vs. float)

Other Errors to Uncover

- Comparison of different data types
- Incorrect logical operators or precedence
- Expectation of equality when precision error makes equality unlikely (using == with float types)
- Incorrect comparison of variables
- Improper or nonexistent loop termination
- Failure to exit when divergent iteration is encountered
- Improperly modified loop variables
- Boundary value violations

Problems to uncover in Error Handling

- Error description is unintelligible or **ambiguous**
- Error noted does not correspond to error encountered
- Error condition causes operating system intervention prior to error handling
- **Exception condition processing** is incorrect
- Error description does not provide enough information to assist in the location of the cause of the error

Drivers and Stubs for Unit Testing

- Driver
 - A simple **main program** that accepts test case data, passes such data to the component being tested, and prints the returned results
- Stubs
 - Serve to replace modules that are subordinate to (called by) the component to be tested
 - It uses the module's exact interface, may do minimal data manipulation, provides verification of entry, and returns control to the module undergoing testing
- Drivers and stubs both represent overhead
 - Both must be written but don't constitute part of the installed software product

**When tester comes by running
and says there is a issue while
release is going on**



Integration Testing

- Defined as a systematic technique for constructing the software architecture
 - At the same time integration is occurring, conduct tests to uncover errors associated with **interfaces**
- Objective is to take unit tested modules and build a program structure based on the prescribed design
- Two Approaches
 - Non-incremental Integration Testing
 - Incremental Integration Testing

Non-incremental Integration Testing

- Commonly called the “Big Bang” approach
- All components are **combined** in advance
- The entire program is tested as a whole
- Chaos results
- Many seemingly-unrelated errors are encountered
- Correction is difficult because isolation of causes is complicated
- Once a set of errors are corrected, more errors occur, and testing appears to enter an endless loop

Incremental Integration Testing

- Three kinds
 - Top-down integration
 - Bottom-up integration
 - Sandwich integration
- The program is constructed and tested in small increments
- Errors are easier to isolate and correct
- Interfaces are more likely to be tested completely
- A systematic test approach is applied

Top-down Integration

- Modules are integrated by moving downward through the control hierarchy, beginning with the main module
- Subordinate modules are incorporated in either a depth-first or breadth-first fashion
 - **DF**: All modules on a major control path are integrated
 - **BF**: All modules directly subordinate at each level are integrated
- **Advantages**
 - This approach verifies **major control or decision points early** in the test process
- **Disadvantages**
 - Stubs need to be created to substitute for modules that have not been built or tested yet; this code is later discarded
 - Because stubs are used to replace lower level modules, no significant data flow can occur until much later in the integration/testing process

Bottom-up Integration

- Integration and testing starts with the most atomic modules in the control hierarchy
- Advantages
 - This approach **verifies low-level data processing** early in the testing process
 - Need for stubs is eliminated
- Disadvantages
 - **Driver modules** need to be built to test the lower-level modules; this code is later discarded or expanded into a full-featured version
 - Drivers inherently do not contain the complete algorithms that will eventually use the services of the lower-level modules; consequently, testing may be incomplete or more testing may be needed later when the upper level modules are available

Bottom-up Integration - example

- **Ex :** Suppose, you are told to test a website whose corresponding primary modules are, where each of them is interdependent on each other, as follows:

Module-A : Login page website,

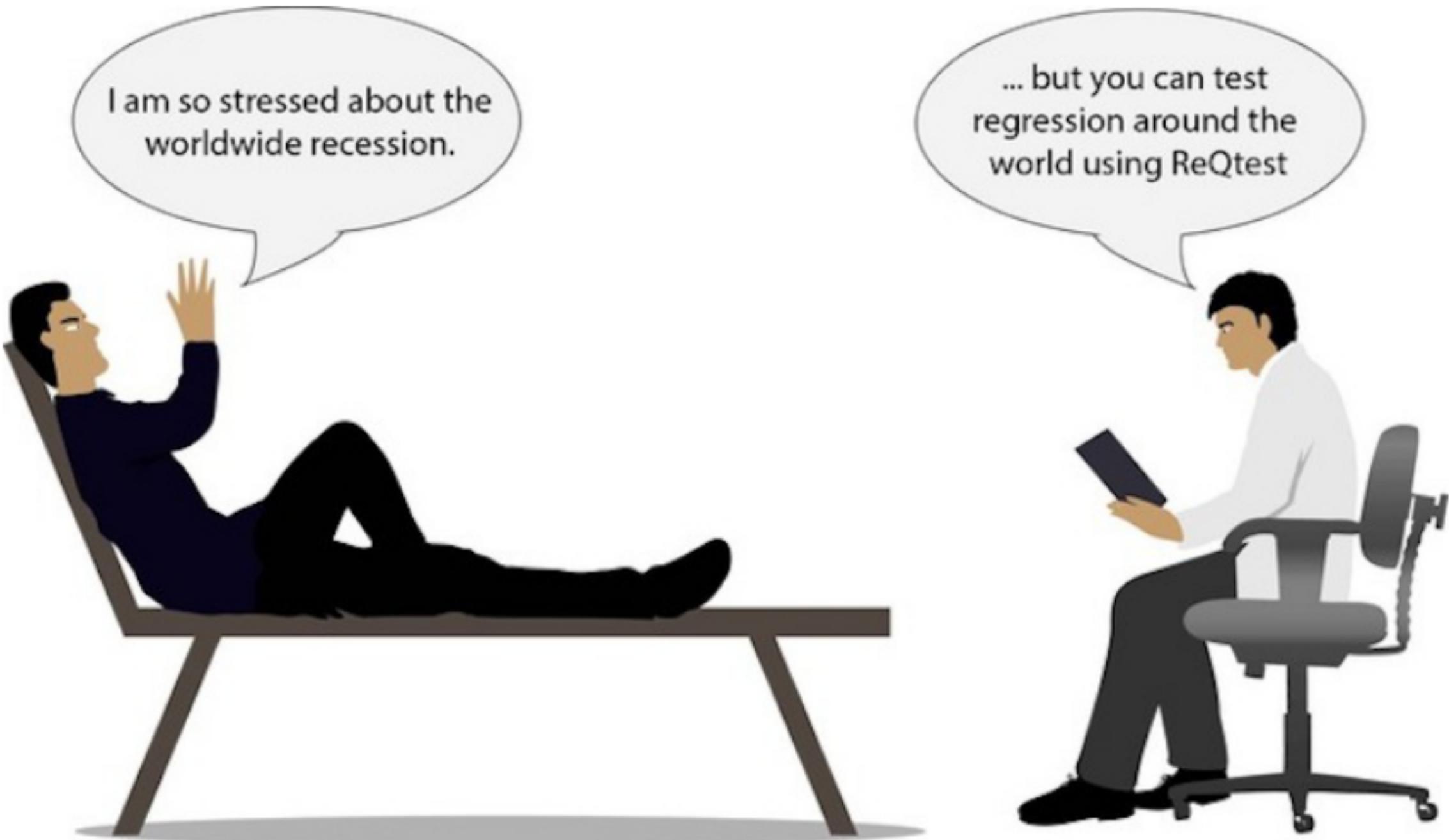
Module-B : Home page of the website

Module-C : Profile setting

Module-D : Sign-out page

Regression Testing

- Each new addition or change to baselined software may cause problems with functions that previously worked flawlessly
- Regression testing re-executes a small subset of tests that have already been conducted
 - Ensures that changes have **not propagated unintended side effects**
 - Helps to ensure that changes do not introduce unintended behavior or additional errors
 - May be done **manually** or through the use of automated capture/ playback tools
- Regression test suite contains three different classes of test cases
 - A representative sample of tests that will exercise all software functions
 - Additional tests that focus on software functions that are likely to be affected by the change
 - Tests that focus on the actual software components that have been changed



Risk Management

- Introduction
- Risk identification
- Risk projection (estimation)
- Risk mitigation, monitoring, and management

Introduction

Definition of Risk

- A risk is a potential problem – it might happen and it might not
- Conceptual definition of risk
 - Risk concerns future happenings
 - Risk involves change in mind, opinion, actions, places, etc.
 - Risk involves choice and the uncertainty that choice entails
- Two characteristics of risk
 - Uncertainty – the risk may or may not happen, that is, there are no 100% risks (those, instead, are called constraints)
 - Loss – the risk becomes a reality and unwanted consequences or losses occur

Risk Categorization – Approach

#1

- Project risks
 - They threaten the project plan
 - If they become real, it is likely that the project schedule will slip and that costs will increase
- Technical risks
 - They threaten the quality and timeliness of the software to be produced
 - If they become real, implementation may become difficult or impossible
- Business risks
 - They threaten the viability of the software to be built
 - If they become real, they jeopardize the project or the product

(More on next slide)

Risk Categorization – Approach #1 (continued)

- Sub-categories of Business risks
 - **Market risk** – building an excellent product or system that no one really wants
 - **Strategic risk** – building a product that no longer fits into the overall business strategy for the company
 - **Sales risk** – building a product that the sales force doesn't understand how to sell
 - **Management risk** – losing the support of senior management due to a change in focus or a change in people
 - **Budget risk** – losing budgetary or personnel commitment

Risk Categorization – Approach #2

- Known risks
 - Those risks that can be uncovered after careful evaluation of the project plan, the business and technical environment in which the project is being developed, and other reliable information sources (e.g., unrealistic delivery date)
- Predictable risks
 - Those risks that are extrapolated from past project experience (e.g., past turnover)
- Unpredictable risks
 - Those risks that can and do occur, but are extremely difficult to identify in advance

Reactive vs. Proactive Risk Strategies

- Reactive risk strategies
 - "Don't worry, I'll think of something"
 - The majority of software teams and managers rely on this approach
 - Nothing is done about risks until something goes wrong
 - The team then flies into action in an attempt to correct the problem rapidly (fire fighting)
 - Crisis management is the choice of management techniques
- Proactive risk strategies
 - Steps for risk management are followed (see next slide)
 - Primary objective is to avoid risk and to have a contingency plan in place to handle unavoidable risks in a controlled and effective manner

Steps for Risk Management

- Identify possible risks; recognize what can go wrong
- Analyze each risk to estimate the probability that it will occur and the impact (i.e., damage) that it will do if it does occur
- Rank the risks by probability and impact
 - Impact may be negligible, marginal, critical, and catastrophic
- Develop a contingency plan to manage those risks having high probability and high impact

Risk Identification

Background

- Risk identification is a systematic attempt to specify threats to the project plan
- By identifying known and predictable risks, the project manager takes a first step toward avoiding them when possible and controlling them when necessary
- Generic risks
 - Risks that are a potential threat to every software project
- Product-specific risks
 - Risks that can be identified only by those with a clear understanding of the technology, the people, and the environment that is specific to the software that is to be built
 - This requires examination of the project plan and the statement of scope
 - "What special characteristics of this product may threaten our project plan?"

Risk Item Checklist

- Used as one way to identify risks
- Focuses on known and predictable risks in specific subcategories (see next slide)
- Can be organized in several ways
 - A list of characteristics relevant to each risk subcategory
 - Questionnaire that leads to an estimate on the impact of each risk
 - A list containing a set of risk component and drivers and their probability of occurrence

Known and Predictable Risk Categories

- **Product size** – risks associated with overall size of the software to be built
- **Business impact** – risks associated with constraints imposed by management or the marketplace
- **Customer characteristics** – risks associated with sophistication of the customer and the developer's ability to communicate with the customer in a timely manner
- **Process definition** – risks associated with the degree to which the software process has been defined and is followed
- **Development environment** – risks associated with availability and quality of the tools to be used to build the project
- **Technology to be built** – risks associated with complexity of the system to be built and the "newness" of the technology in the system
- **Staff size and experience** – risks associated with overall technical and project experience of the software engineers who will do the work

Questionnaire on Project Risk

(Questions are ordered by their relative importance to project success)

- Have top software and customer managers formally committed to support the project?
- Are end-users enthusiastically committed to the project and the system/product to be built?
- Are requirements fully understood by the software engineering team and its customers?
- Have customers been involved fully in the definition of requirements?
- Do end-users have realistic expectations?
- Is the project scope stable?

Questionnaire on Project Risk (continued)

- Does the software engineering team have the right mix of skills?
- Are project requirements stable?
- Does the project team have experience with the technology to be implemented?
- Is the number of people on the project team adequate to do the job?
- Do all customer/user constituencies agree on the importance of the project and on the requirements for the system/product to be built?

Risk Components and Drivers

- The project manager identifies the risk drivers that affect the following risk components
 - **Performance risk** - the degree of uncertainty that the product will meet its requirements and be fit for its intended use
 - **Cost risk** - the degree of uncertainty that the project budget will be maintained
 - **Support risk** - the degree of uncertainty that the resultant software will be easy to correct, adapt, and enhance
 - **Schedule risk** - the degree of uncertainty that the project schedule will be maintained and that the product will be delivered on time
- The impact of each risk driver on the risk component is divided into one of four impact levels
 - Negligible, marginal, critical, and catastrophic
- Risk drivers can be assessed as impossible, improbable, probable, and frequent

Risk Projection (Estimation)

Background

- Risk projection (or estimation) attempts to rate each risk in two ways
 - The probability that the risk is real
 - The consequence of the problems associated with the risk, should it occur
- The project planner, managers, and technical staff perform four risk projection steps (see next slide)
- The intent of these steps is to consider risks in a manner that leads to prioritization
- By prioritizing risks, the software team can allocate limited resources where they will have the most impact

Risk Projection/Estimation Steps

- Establish a scale that reflects the perceived likelihood of a risk (e.g., 1-low, 10-high)
- Delineate the consequences of the risk
- Estimate the impact of the risk on the project and product
- Note the overall accuracy of the risk projection so that there will be no misunderstandings

Contents of a Risk Table

- A risk table provides a project manager with a simple technique for risk projection
- It consists of five columns
 - Risk Summary – short description of the risk
 - Risk Category – one of seven risk categories (slide 12)
 - Probability – estimation of risk occurrence based on group input
 - Impact – (1) catastrophic (2) critical (3) marginal (4) negligible
 - RMMM – Pointer to a paragraph in the Risk Mitigation, Monitoring, and Management Plan

Risk Summary	Risk	Probability	Impact (1-4)	RMMM
	Category			

(More on next slide)

Developing a Risk Table

- List all risks in the first column (by way of the help of the risk item checklists)
- Mark the category of each risk
- Estimate the probability of each risk occurring
- Assess the impact of each risk based on an averaging of the four risk components to determine an overall impact value (See next slide)
- Sort the rows by probability and impact in descending order
- Draw a horizontal cutoff line in the table that indicates the risks that will be given further attention

Assessing Risk Impact

- Three factors affect the consequences that are likely if a risk does occur
 - Its nature – This indicates the problems that are likely if the risk occurs
 - Its scope – This combines the severity of the risk (how serious was it) with its overall distribution (how much was affected)
 - Its timing – This considers when and for how long the impact will be felt
- The overall risk exposure formula is $RE = P \times C$
 - P = the probability of occurrence for a risk
 - C = the cost to the project should the risk actually occur
- Example
 - P = 80% probability that 18 of 60 software components will have to be developed
 - C = Total cost of developing 18 components is \$25,000
 - $RE = .80 \times \$25,000 = \$20,000$

Risk Mitigation, Monitoring, and Management

Background

- An effective strategy for dealing with risk must consider three issues
 - (Note: these are not mutually exclusive)
 - Risk mitigation (i.e., avoidance)
 - Risk monitoring
 - Risk management and contingency planning
- Risk mitigation (avoidance) is the primary strategy and is achieved through a plan
 - Example: Risk of high staff turnover (see next slide)

(More on next slide)

23

Background (continued)

Strategy for Reducing Staff Turnover

- Meet with current staff to determine causes for turnover (e.g., poor working conditions, low pay, competitive job market)
- Mitigate those causes that are under our control before the project starts
- Once the project commences, assume turnover will occur and develop techniques to ensure continuity when people leave
- Organize project teams so that information about each development activity is widely dispersed
- Define documentation standards and establish mechanisms to ensure that documents are developed in a timely manner
- Conduct peer reviews of all work (so that more than one person is "up to speed")
- Assign a backup staff member for every critical technologist

Background (continued)

- During risk monitoring, the project manager monitors factors that may provide an indication of whether a risk is becoming more or less likely
- Risk management and contingency planning assume that mitigation efforts have failed and that the risk has become a reality
- RMMM steps incur additional project cost
 - Large projects may have identified 30 – 40 risks
- Risk is not limited to the software project itself
 - Risks can occur after the software has been delivered to the user

(More on next slide)

Background (continued)

- Software safety and hazard analysis
 - These are software quality assurance activities that focus on the identification and assessment of potential hazards that may affect software negatively and cause an entire system to fail
 - If hazards can be identified early in the software process, software design features can be specified that will either eliminate or control potential hazards

The RMMM Plan

- The RMMM plan may be a part of the software development plan (Paragraph 5.19.1) or may be a separate document
- Once RMMM has been documented and the project has begun, the risk mitigation, and monitoring steps begin
 - Risk mitigation is a problem avoidance activity
 - Risk monitoring is a project tracking activity
- Risk monitoring has three objectives
 - To assess whether predicted risks do, in fact, occur
 - To ensure that risk aversion steps defined for the risk are being properly applied
 - To collect information that can be used for future risk analysis
- The findings from risk monitoring may allow the project manager to ascertain what risks caused which problems throughout the project

Seven Principles of Risk Management

- **Maintain a global perspective**
 - View software risks within the context of a system and the business problem that it is intended to solve
- **Take a forward-looking view**
 - Think about risks that may arise in the future; establish contingency plans
- **Encourage open communication**
 - Encourage all stakeholders and users to point out risks at any time
- **Integrate risk management**
 - Integrate the consideration of risk into the software process
- **Emphasize a continuous process of risk management**
 - Modify identified risks as more becomes known and add new risks as better insight is achieved
- **Develop a shared product vision**
 - A shared vision by all stakeholders facilitates better risk identification and assessment
- **Encourage teamwork when managing risk**
 - Pool the skills and experience of all stakeholders when conducting risk management activities

Summary

- Whenever much is riding on a software project, common sense dictates risk analysis
 - Yet, most project managers do it informally and superficially, if at all
- However, the time spent in risk management results in
 - Less upheaval during the project
 - A greater ability to track and control a project
 - The confidence that comes with planning for problems before they occur
- Risk management can absorb a significant amount of the project planning effort...but the effort is worth it



A time of Gratitude

- Almighty God in protecting in this uncertain times.
- Your sincerity and dedication in learning and cooperation with me in this awesome course
- Your great efforts and passion to explore the concepts in SE and active participation in the class room
- For your family members
- You all proved that

simply ^{so} GREAT!

Appreciations

- Tough questions makes teaching interesting!

Token of appreciations for their active interaction in the classes (by way of questioning and answering)

Preparing for End Sem

HOW... WHAT to Learn?

- Read the standard materials,
 - Text books,
 - My slides
 - NPTEL videos
-
- Have a planned schedule and focus on understanding the fundamental concepts,
 - Practice questions from,
 - Book backs
 - GATE questions
 - coding for exercises in book backs, standard institute's assignment/exam questions ³²

Extra Miles...

- Do Online coding,
- Summer/Internship
- Thon- events
- Open project/Innovation competitions
- Develop a website of your own – update achievements, solved problems with their codes
- Build a CV first!



Love SE always! ☺, there is a better career...

Keep in Touch

Email : oswald@nitt.edu
oswald.mecse@gmail.com

Software Project Scheduling

- Introduction
- Project scheduling
- Task network
- Timeline chart
- Earned value analysis

Introduction

Eight Reasons for Late Software Delivery

- An unrealistic deadline established by someone outside the software engineering group and forced on managers and practitioners within the group
- Changing customer requirements that are not reflected in schedule changes
- An honest underestimate of the amount of effort and /or the number of resources that will be required to do the job
- Predictable and/or unpredictable risks that were not considered when the project commenced
- Technical difficulties that could not have been foreseen in advance
- Human difficulties that could not have been foreseen in advance
- Miscommunication among project staff that results in delays
- A failure by project management to recognize that the project is falling behind schedule and a lack of action to correct the problem

Quote from Napoleon

"Any commander-in-chief who undertakes to carry out a plan which he considers defective is at fault; he must put forth his reasons, insist on the plan being changed, and finally tender his resignation rather than be the instrument of his army's downfall."

I'LL NEED A PROJECT
PLAN TO JUSTIFY
THE RESOURCES WE
NEED TO CHANGE OUR
SOFTWARE.

I CAN MAKE THOSE
SOFTWARE CHANGES
IN TEN SECONDS.

DONE.

GOOD WORK.
NOW ALL WE
NEED IS THAT
PLAN.

www.dilbert.com

scottadams@aol.com

11/10/98 © 1998 United Feature Syndicate, Inc.

Handling Unrealistic Deadlines

- Perform a detailed estimate using historical data from past projects; determine the estimated effort and duration for the project
- Using an incremental model, develop a software engineering strategy that will deliver critical functionality by the imposed deadline, but delay other functionality until later; document the plan
- Meet with the customer and (using the detailed estimate) explain why the imposed deadline is unrealistic
 - Be certain to note that all estimates are based on performance on past projects
 - Also be certain to indicate the percent improvement that would be required to achieve the deadline as it currently exists
- Offer the incremental development strategy as an alternative and offer some options
 - Increase the budget and bring on additional resources to try to finish sooner
 - Remove many of the software functions and capabilities that were requested
 - Dispense with reality and wish the project complete using the prescribed schedule; then point out that project history and your estimates show that this is unrealistic and will result in a disaster

Project Scheduling

General Practices

- On large projects, hundreds of small tasks must occur to accomplish a larger goal
 - Some of these tasks lie outside the mainstream and may be completed without worry of impacting on the project completion date
 - Other tasks lie on the critical path; if these tasks fall behind schedule, the completion date of the entire project is put into jeopardy
- Project manager's objectives
 - Define all project tasks
 - Build an activity network that depicts their interdependencies
 - Identify the tasks that are critical within the activity network
 - Build a timeline depicting the planned and actual progress of each task
 - Track task progress to ensure that delay is recognized "one day at a time"
 - To do this, the schedule should allow progress to be monitored and the project to be controlled

General Practices (continued)

- Software project scheduling distributes estimated effort across the planned project duration by allocating the effort to specific tasks
- During early stages of project planning, a macroscopic schedule is developed identifying all major process framework activities and the product functions to which they apply
- Later, each task is refined into a detailed schedule where specific software tasks are identified and scheduled
- Scheduling for projects can be viewed from two different perspectives
 - In the first view, an end-date for release of a computer-based system has already been established and fixed
 - The software organization is constrained to distribute effort within the prescribed time frame
 - In the second view, assume that rough chronological bounds have been discussed but that the end-date is set by the software engineering organization
 - Effort is distributed to make best use of resources and an end-date is defined after careful analysis of the software

Basic Principles for Project Scheduling

- Compartmentalization
 - The project must be compartmentalized into a number of manageable activities, actions, and tasks; both the product and the process are decomposed
- Interdependency
 - The interdependency of each compartmentalized activity, action, or task must be determined
 - Some tasks must occur in sequence while others can occur in parallel
 - Some actions or activities cannot commence until the work product produced by another is available
- Time allocation
 - Each task to be scheduled must be allocated some number of work units
 - In addition, each task must be assigned a start date and a completion date that are a function of the interdependencies
 - Start and stop dates are also established based on whether work will be conducted on a full-time or part-time basis
(More on next slide)

Basic Principles for Project Scheduling (continued)

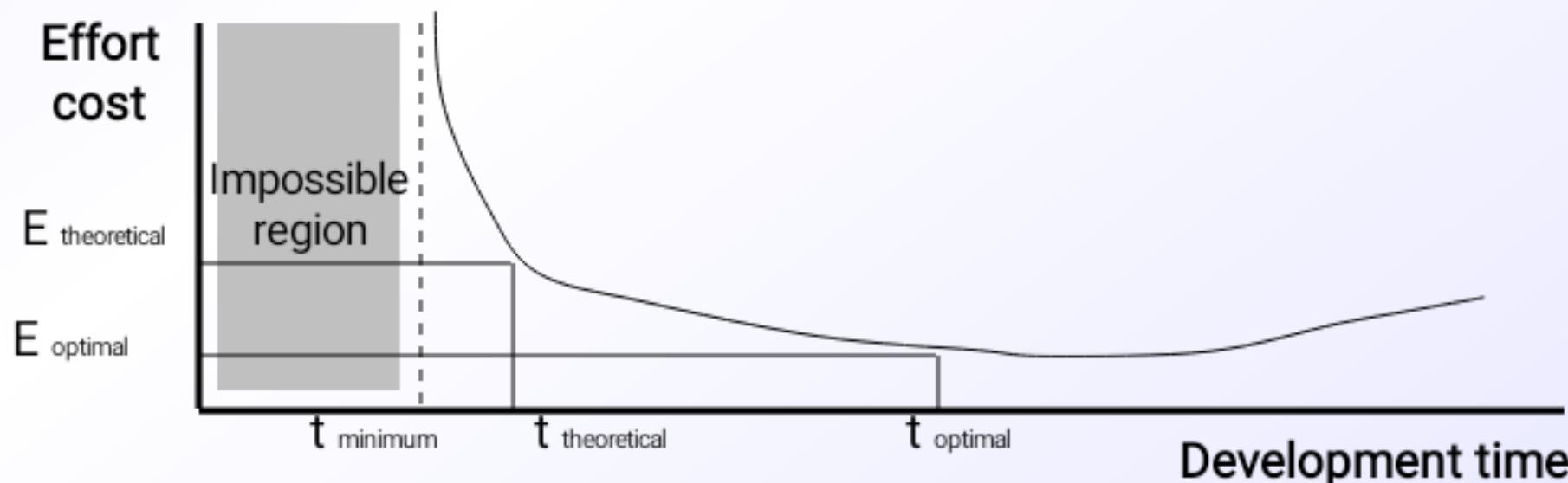
- Effort validation
 - Every project has a defined number of people on the team
 - As time allocation occurs, the project manager must ensure that no more than the allocated number of people have been scheduled at any given time
- Defined responsibilities
 - Every task that is scheduled should be assigned to a specific team member
- Defined outcomes
 - Every task that is scheduled should have a defined outcome for software projects such as a work product or part of a work product
 - Work products are often combined in deliverables
- Defined milestones
 - Every task or group of tasks should be associated with a project milestone
 - A milestone is accomplished when one or more work products has been reviewed for quality and has been approved

Relationship Between People and Effort

- Common management myth: *If we fall behind schedule, we can always add more programmers and catch up later in the project*
 - This practice actually has a disruptive effect and causes the schedule to slip even further
 - The added people must learn the system
 - The people who teach them are the same people who were earlier doing the work
 - During teaching, no work is being accomplished
 - Lines of communication (and the inherent delays) increase for each new person added

Effort Applied vs. Delivery Time

- There is a nonlinear relationship between effort applied and delivery time (Ref: Putnam-Norden-Rayleigh Curve)
 - Effort increases rapidly as the delivery time is reduced
- Also, delaying project delivery can reduce costs significantly as shown in the equation $E = L^3/(P^3 t^4)$ and in the curve below
 - E = development effort in person-months
 - L = source lines of code delivered
 - P = productivity parameter (ranging from 2000 to 12000)
 - t = project duration in calendar months



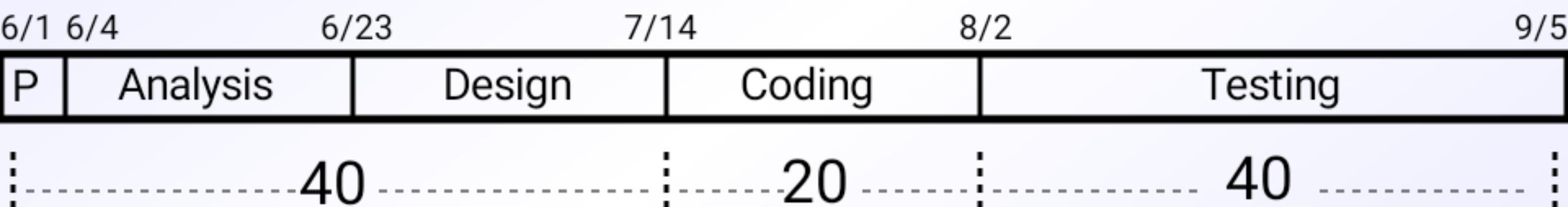
40-20-40 Distribution of Effort

- A recommended distribution of effort across the software process is 40% (analysis and design), 20% (coding), and 40% (testing)
- Work expended on project planning rarely accounts for more than 2 - 3% of the total effort
- Requirements analysis may comprise 10 - 25%
 - Effort spent on prototyping and project complexity may increase this
- Software design normally needs 20 – 25%
- Coding should need only 15 - 20% based on the effort applied to software design
- Testing and subsequent debugging can account for 30 - 40%
 - Safety or security-related software requires more time for testing

(More on next slide)

40-20-40 Distribution of Effort (continued)

Example: 100-day project



Task Network

Defining a Task Set

- A task set is the work breakdown structure for the project
- No single task set is appropriate for all projects and process models
 - It varies depending on the project type and the degree of rigor (based on influential factors) with which the team plans to work
- The task set should provide enough discipline to achieve high software quality
 - But it must not burden the project team with unnecessary work

Types of Software Projects

- Concept development projects
 - Explore some new business concept or application of some new technology
- New application development
 - Undertaken as a consequence of a specific customer request
- Application enhancement
 - Occur when existing software undergoes major modifications to function, performance, or interfaces that are observable by the end user
- Application maintenance
 - Correct, adapt, or extend existing software in ways that may not be immediately obvious to the end user
- Reengineering projects
 - Undertaken with the intent of rebuilding an existing (legacy) system in whole or in part

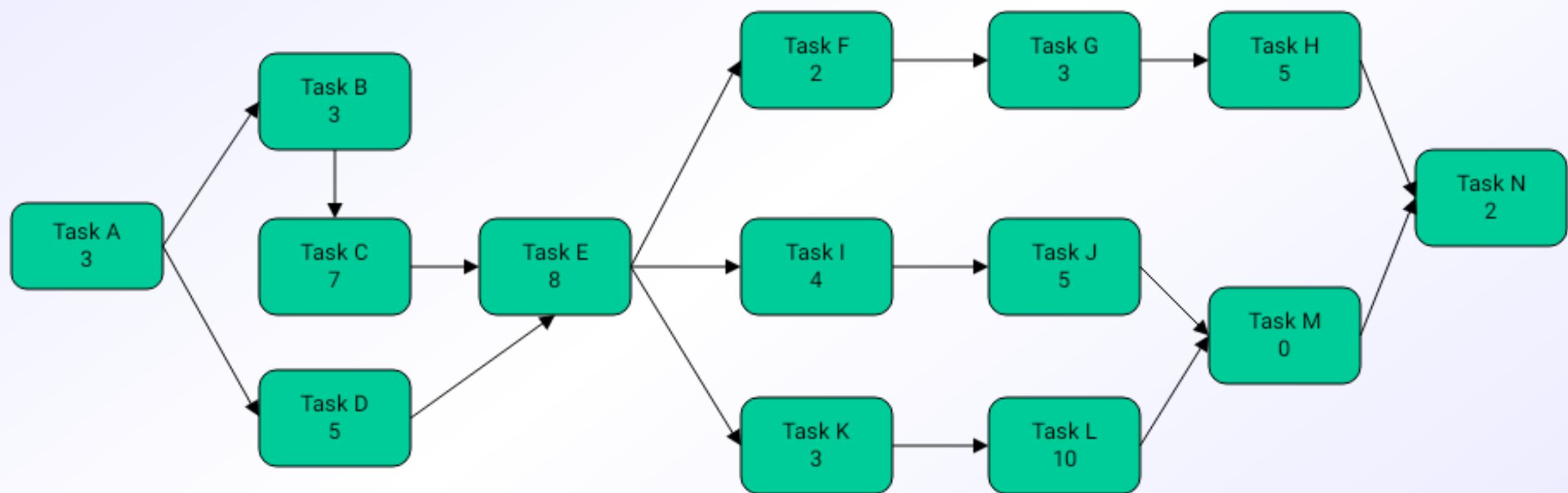
Factors that Influence a Project's Schedule

- Size of the project
- Number of potential users
- Mission criticality
- Application longevity
- Stability of requirements
- Ease of customer/developer communication
- Maturity of applicable technology
- Performance constraints
- Embedded and non-embedded characteristics
- Project staff
- Reengineering factors

Purpose of a Task Network

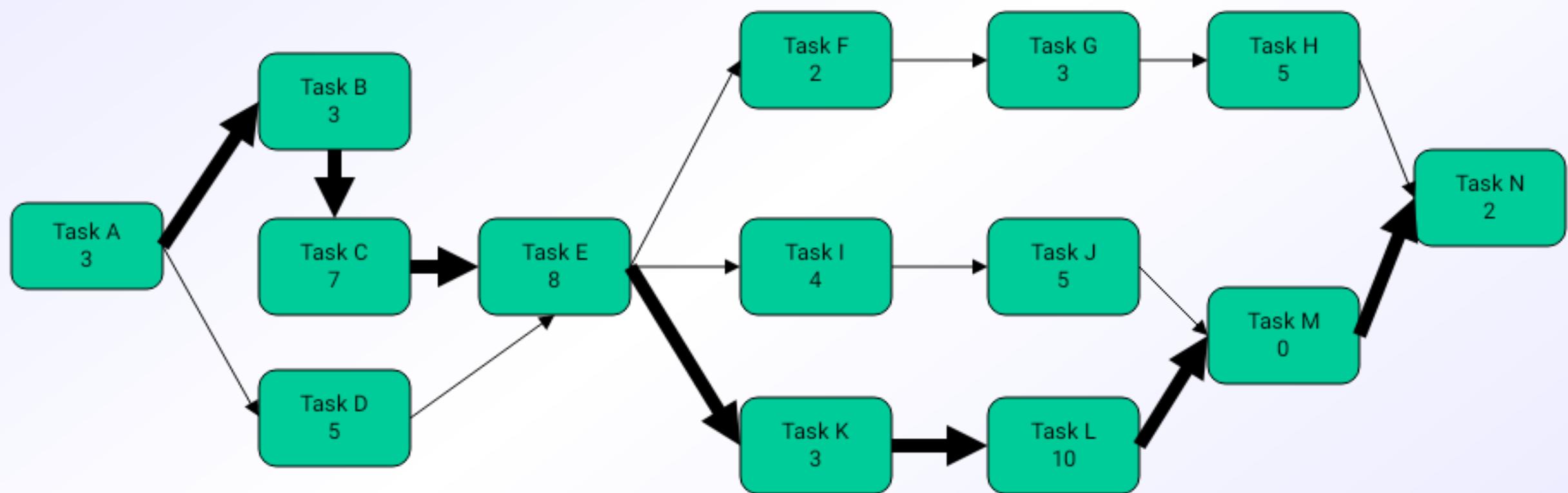
- Also called an activity network
- It is a graphic representation of the task flow for a project
- It depicts task length, sequence, concurrency, and dependency
- Points out inter-task dependencies to help the manager ensure continuous progress toward project completion
- The critical path
 - A single path leading from start to finish in a task network with the sum of the task completion time being maximum.
 - It contains the sequence of tasks that must be completed on schedule if the project as a whole is to be completed on schedule
 - It also determines the minimum duration of the project – how?

Example Task Network



Where is the critical path and what tasks are on it?

Example Task Network with Critical Path Marked



Critical path: A-B-C-E-K-L-M-N

Timeline Chart

Mechanics of a Timeline Chart

- Also called a **Gantt chart**; invented by Henry Gantt, industrial engineer, 1917
 - All project tasks are listed in the far left column
 - The next few columns may list the following for each task: projected start date, projected stop date, projected duration, actual start date, actual stop date, actual duration, task inter-dependencies (i.e., predecessors)
 - To the far right are columns representing dates on a calendar
 - The length of a horizontal bar on the calendar indicates the duration of the task
 - When multiple bars occur at the same time interval on the calendar, this implies task concurrency
 - A diamond in the calendar area of a specific task indicates that the task is a milestone; a milestone has a time duration of zero

Task #	Task Name	Duration	Start	Finish	Pred.									
1	Task A	2 months	1/1	2/28	None									
2	Milestone N	0	3/1	3/1	1									

CLASS EXERCISE

Timeline chart:

4/1 4/8 4/15 4/22 4/29 5/6 5/13 5/20 5/27 6/3

Task #	Task Name	Duration (days)	Start	Finish	Pred.								
A	Establish increments	3	4/1		None								
B	Analyze Inc One	3			A								
C	Design Inc One	8			B								
D	Code Inc One	7			C								
E	Test Inc One	10			D								
F	Install Inc One	5			E								
G	Analyze Inc Two	7			A, B								
H	Design Inc Two	5			G								
I	Code Inc Two	4			H								
J	Test Inc Two	6			E, I								
K	Install Inc Two	2			J								
L	Close out project	2			F, K								

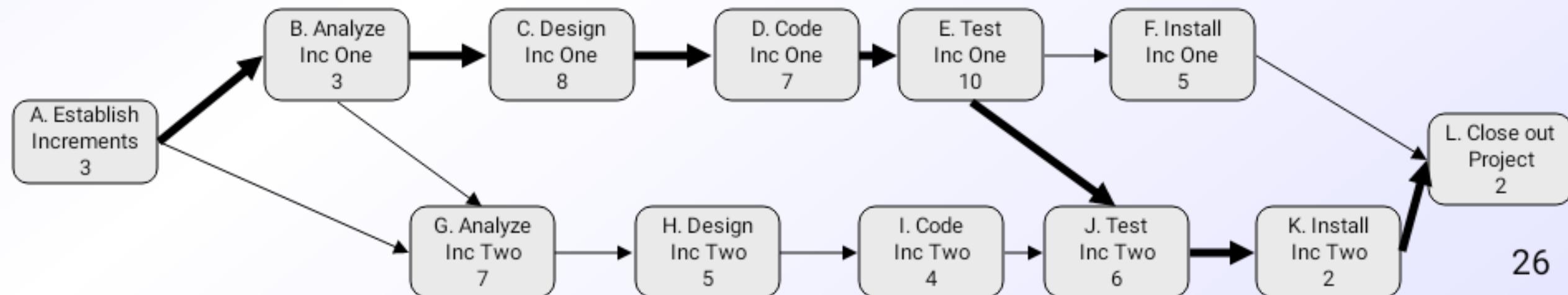
Task network and the critical path:

SOLUTION

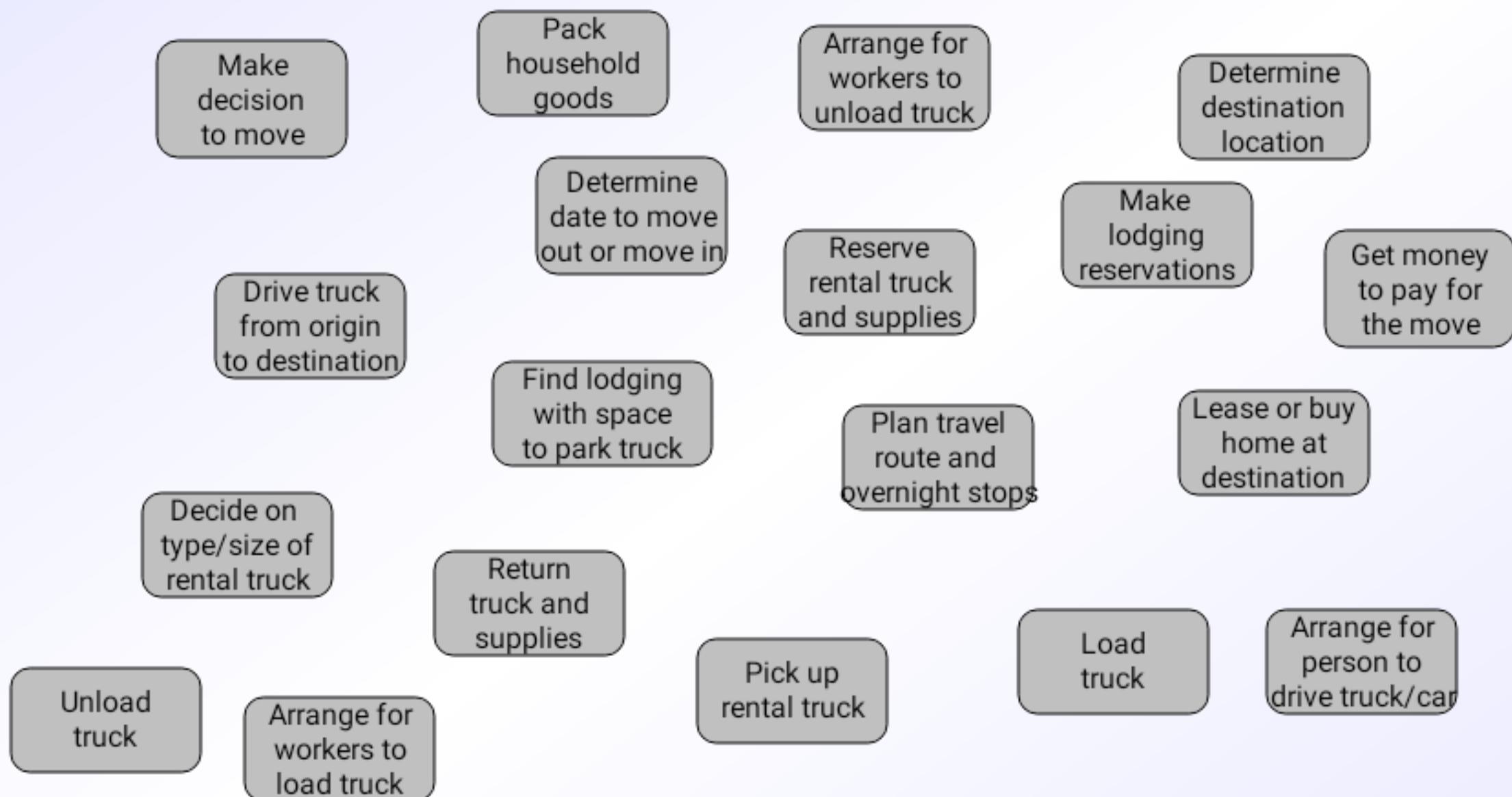
Timeline chart:

Task #	Task Name	Duration	Start	Finish	Pred.	4/1	4/8	4/15	4/22	4/29	5/6	5/13	5/20	5/27	6/3	
A	Establish increments	3	4/1	4/3	None											
B	Analyze Inc One	3	4/4	4/6	A											
C	Design Inc One	8	4/7	4/14	B											
D	Code Inc One	7	4/15	4/21	C											
E	Test Inc One	10	4/22	5/1	D											
F	Install Inc One	5	5/2	5/6	E											
G	Analyze Inc Two	7	4/7	4/13	A, B											
H	Design Inc Two	5	4/14	4/18	G											
I	Code Inc Two	4	4/19	4/22	H											
J	Test Inc Two	6	5/2	5/7	E, I											
K	Install Inc Two	2	5/8	5/9	J											
L	Close out project	2	5/10	5/11	F, K											

Task network and the critical path A-B-C-D-E-J-K-L

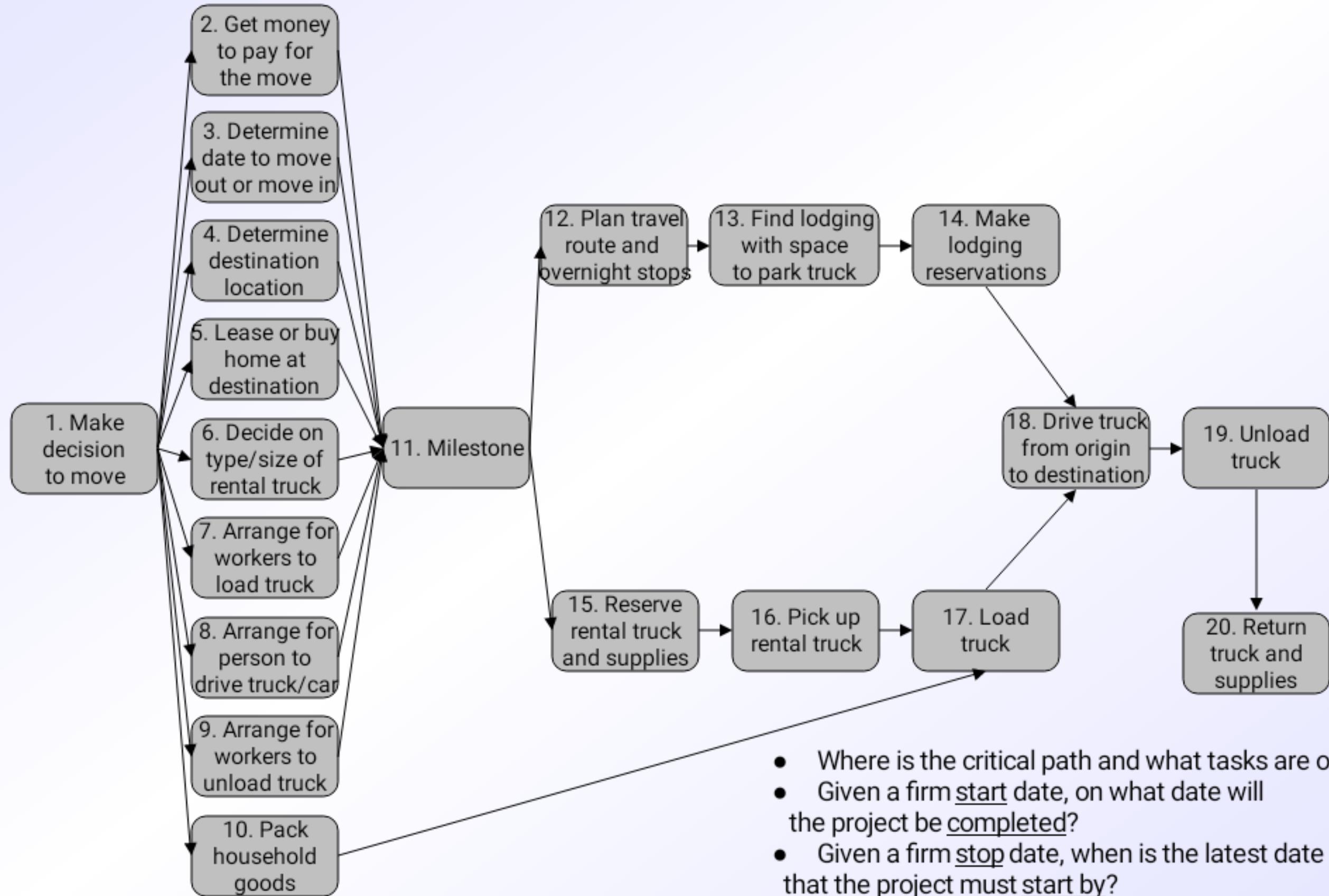


Proposed Tasks for a Long-Distance Move of 8,000 lbs of Household Goods



- Where is the critical path and what tasks are on it?
- Given a firm start date, on what date will the project be completed?
- Given a firm stop date, when is the latest date that the project must start by?

Task Network for a Long-Distance Move of 8,000 lbs of Household Goods



- Where is the critical path and what tasks are on it?
- Given a firm start date, on what date will the project be completed?
- Given a firm stop date, when is the latest date that the project must start by?

Timeline Chart for Long Distance Move

Example Timeline Chart

For this particular project, the Gantt chart was useful mainly for tracking progress and visualizing how much time is left for each stage. Excel was chosen as the medium for developing the Gantt chart because all members had access to Excel and it was fairly easy to update or change without requiring any HTML coding or similar methods.

Task Analysis Group Project		Winter 2001				Updated:01.02.05									
						Light shade = Proposed		Dark shade = Actual		xxxxxxxx = Milestone					
		Jan. 9	Jan. 16	Jan. 23	Jan. 30	Feb. 6	Feb. 13	Feb. 20	Feb. 27	Mar. 6	Mar. 13	Mar. 20	Mar. 27	Apr. 3	Apr. 10
1.0 Learner Profiles								XXXXXXXXXX							
1.1 Talk with project advisor						XXXXXXXXXX									
1.2 Write up profile						XXXXXXXXXX									
2.0 Design						XXXXXXXXXX									
2.1 Brainstorm Ideas						XXXXXXXXXX									
2.2 Choose content and design concept						XXXXXXXXXX									
2.3 Develop Story Boards - paper						XXXXXXXXXX				XXXXXXXXXXXX					
2.4 Review Story Boards with advisor						XXXXXXXXXX				XXXXXXXXXXXX					
3.0 Prototype										XXXXXXXXXX					
3.1 Find/prepare graphics/content										XXXXXXXXXX					
3.2 Code interface										XXXXXXXXXX					
3.3 Test/debug interface										XXXXXXXXXX					
3.4 Review prototype with advisor										XXXXXXXXXX					
4.0 Evaluation Process											XXXXXXXXXXXX				
4.1 Determine what to evaluate											XXXXXXXXXX				
4.2 Evaluation environment											XXXXXXXXXX				
4.3 Determine length of time											XXXXXXXXXX				
4.4 Conduct evaluation											XXXXXXXXXX				
5.0 Results of Evaluation												XXXXXXXXXXXX			
5.1 Analyze result												XXXXXXXXXX			
5.2 Write up results												XXXXXXXXXX			
5.3 Recommend design changes												XXXXXXXXXX			
5.4 Present to client												XXXXXXXXXXXX			
6.0 Design Rational Web Site													XXXXXXXXXX		XXXXXXXXXXXX

Methods for Tracking the Schedule

- Qualitative approaches
 - Conduct periodic project status meetings in which each team member reports progress and problems
 - Evaluate the results of all reviews conducted throughout the software engineering process
 - Determine whether formal project milestones (i.e., diamonds) have been accomplished by the scheduled date
 - Compare actual start date to planned start date for each project task listed in the timeline chart
 - Meet informally with the software engineering team to obtain their subjective assessment of progress to date and problems on the horizon
- Quantitative approach
 - Use earned value analysis to assess progress quantitatively

"The basic rule of software status reporting can be summarized in a single phrase: No surprises." Capers Jones

Project Control and Time Boxing

- The project manager applies control to administer project resources, cope with problems, and direct project staff
- If things are going well (i.e., schedule, budget, progress, milestones) then control should be light
- When problems occur, the project manager must apply tight control to reconcile the problems as quickly as possible. For example:
 - Staff may be redeployed
 - The project schedule may be redefined
- Severe deadline pressure may require the use of time boxing
 - An incremental software process is applied to the project
 - The tasks associated with each increment are “time-boxed” (i.e., given a specific start and stop time) by working backward from the delivery date
 - The project is not allowed to get “stuck” on a task
 - When the work on a task hits the stop time of its box, then work ceases on that task and the next task begins
 - This approach succeeds based on the premise that when the time-box boundary is encountered, it is likely that 90% of the work is complete
 - The remaining 10% of the work can be

Milestones for OO Projects

- Task parallelism in object-oriented projects makes project tracking more difficult to do than non-OO projects because a number of different activities can be happening at once
- Sample milestones
 - Object-oriented analysis completed
 - Object-oriented design completed
 - Object-oriented coding completed
 - Object-oriented testing completed
- Because the object-oriented process is an iterative process, each of these milestones may be revisited as different increments are delivered to the customer

Earned Value Analysis

Description of Earned Value Analysis

- Earned value analysis is a measure of progress by assessing the percent of completeness for a project
- It gives accurate and reliable readings of performance very early into a project
- It provides a common value scale (i.e., time) for every project task, regardless of the type of work being performed
- The total hours to do the whole project are estimated, and every task is given an earned value based on its estimated percentage of the total

Determining Earned Value

- Compute the budgeted cost of work scheduled (BCWS) for each work task i in the schedule
 - The BCWS is the effort planned; work is estimated in person-hours or person-days for each task
 - To determine progress at a given point along the project schedule, the value of BCWS is the sum of the BCWS $_i$ values of all the work tasks that should have been completed by that point of time in the project schedule
- Sum up the BCWS values for all work tasks to derive the budget at completion (BAC)
- Compute the value for the budgeted cost of work performed (BCWP)
 - BCWP is the sum of the BCWS values for all work tasks that have actually been completed by a point of time on the project schedule³⁶

Progress Indicators provided through Earned Value Analysis

- SPI = BCWP/BCWS
 - Schedule performance index (SPI) is an indication of the efficiency with which the project is utilizing scheduled resources
 - SPI close to 1.0 indicates efficient execution of the project schedule
- SV = BCWP – BCWS
 - Schedule variance (SV) is an absolute indication of variance from the planned schedule

