



Software Engineering (CSPE41)

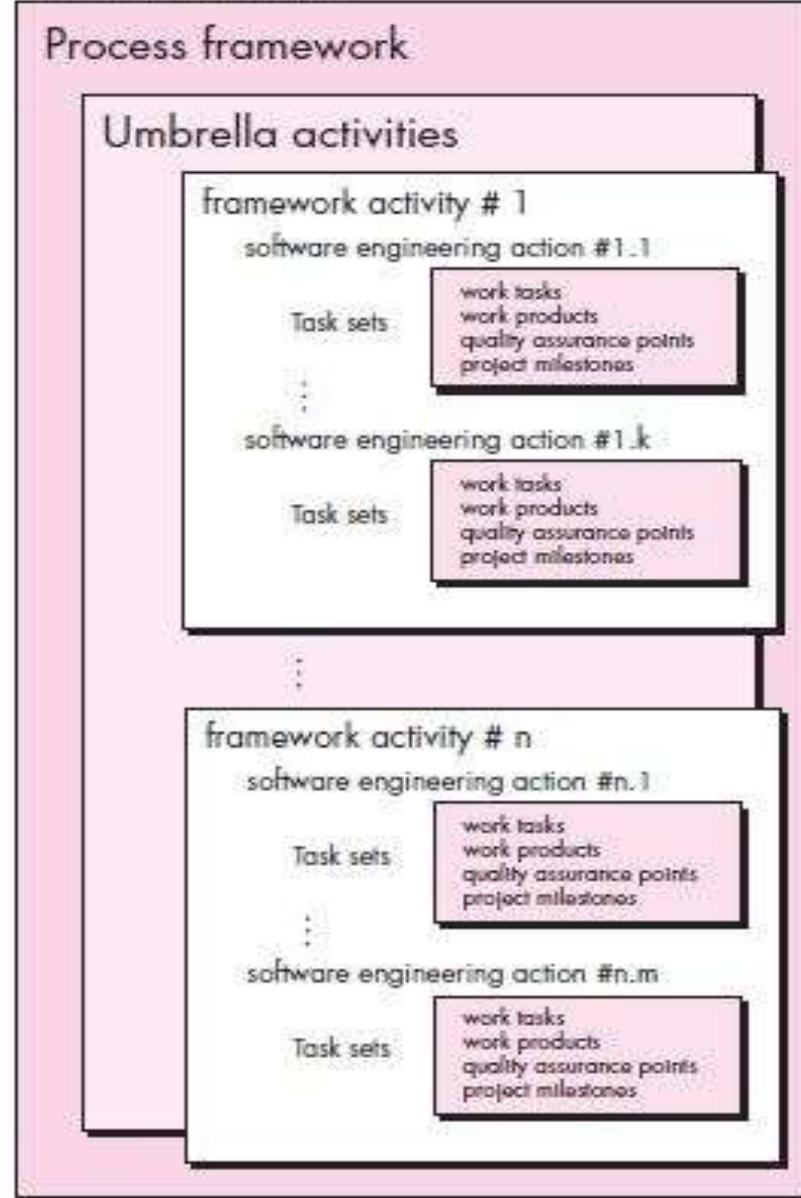
Summer 2023

Lecture 2,3, 4 and 5:
Software Development Life Cycle (SDLC) Models

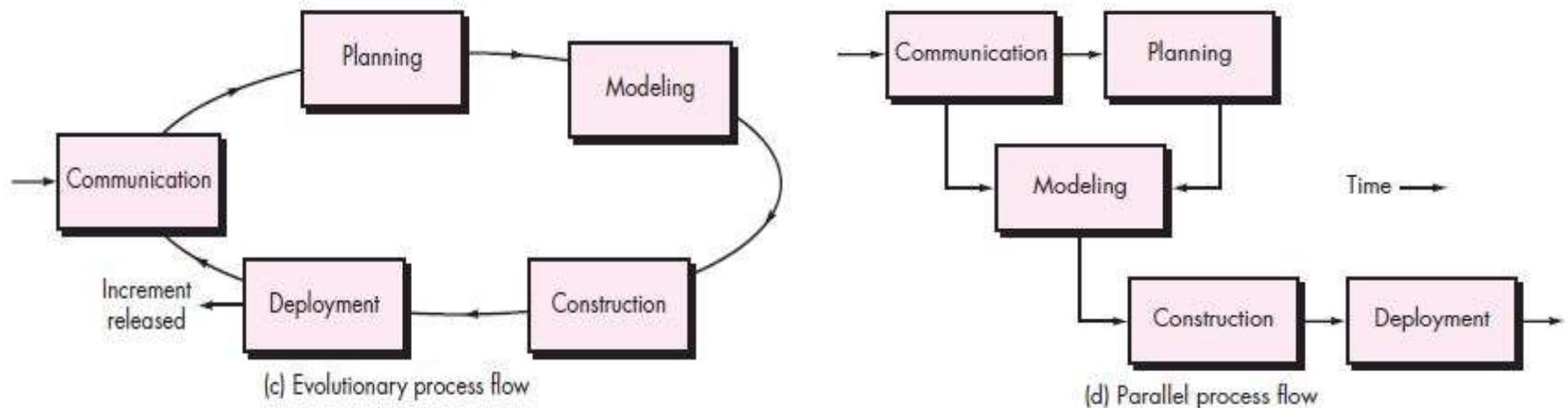
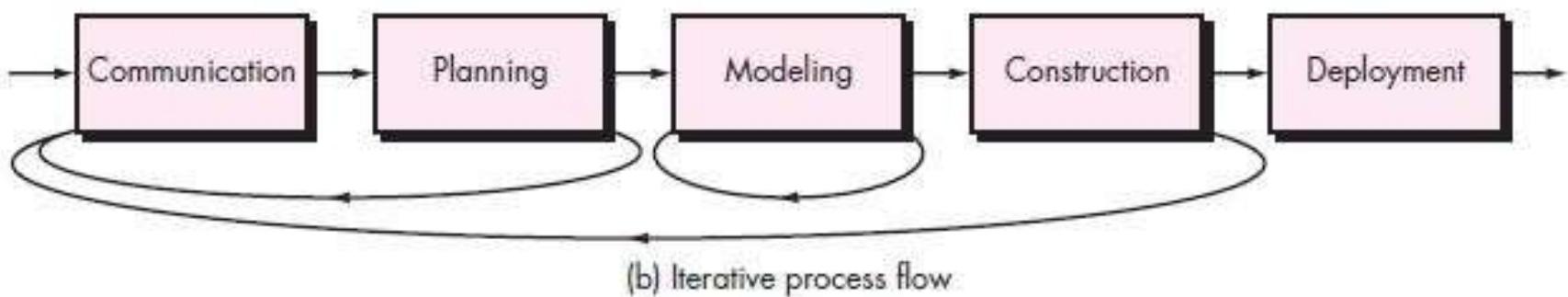
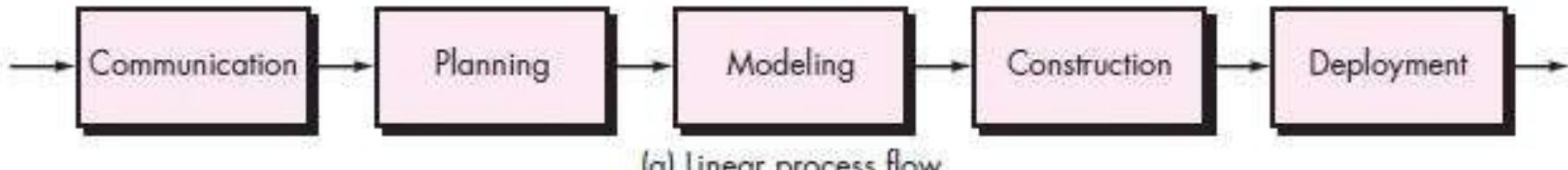
Slides based on various web sources including Pressman's text

A Generic Process Model

Software process



Process Flows



Identifying a Task Set

- A task set defines the actual work to be done to accomplish the objectives of a software engineering action.
 - A list of the tasks to be accomplished
 - A list of the work products to be produced
 - A list of the quality assurance filters to be applied

Software Development Life Cycle (SDLC)

“You’ve got to be very careful if you don’t know where you’re going, because you might not get there.”

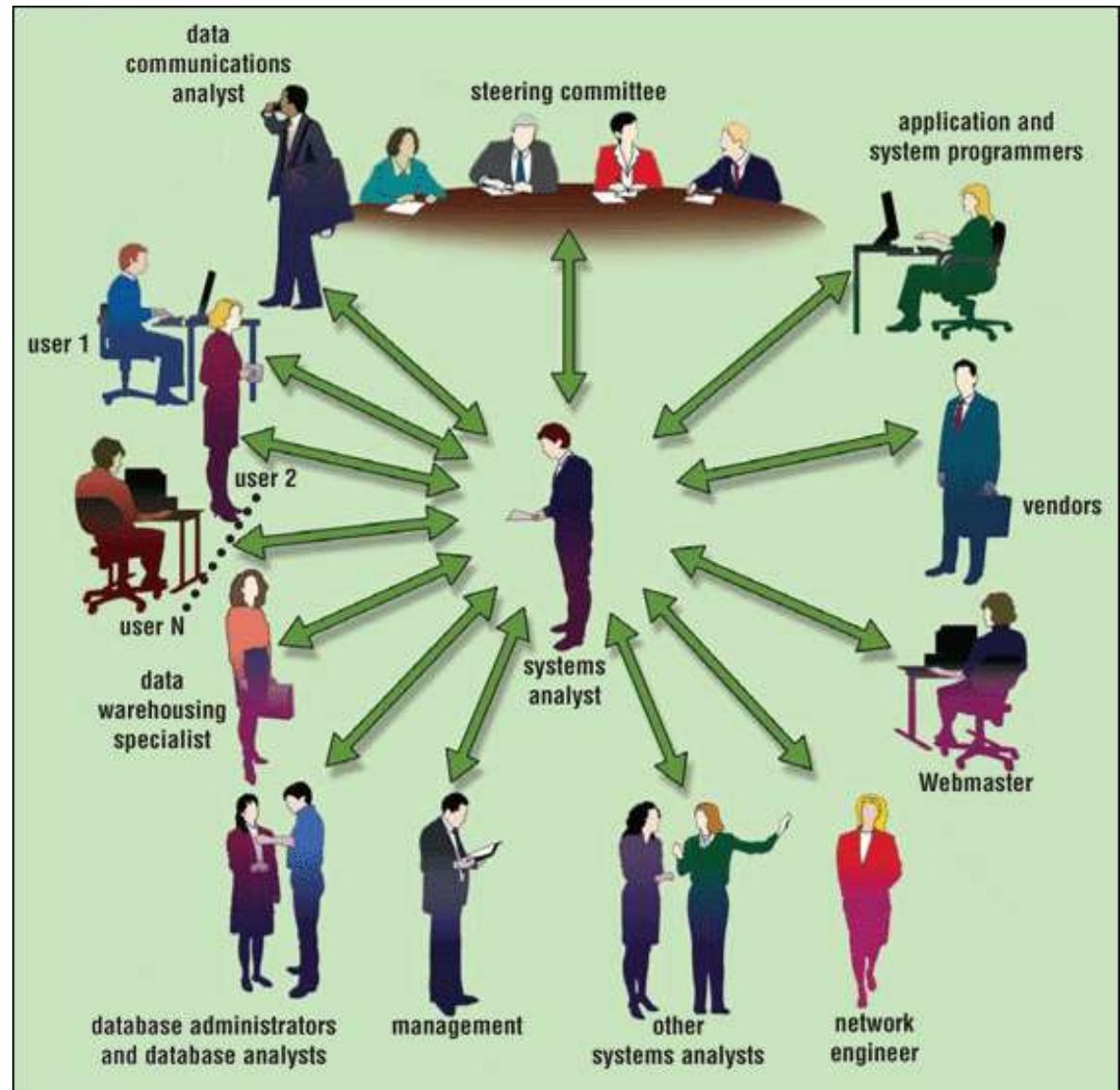
Yogi Berra

Software Development Life Cycle Model

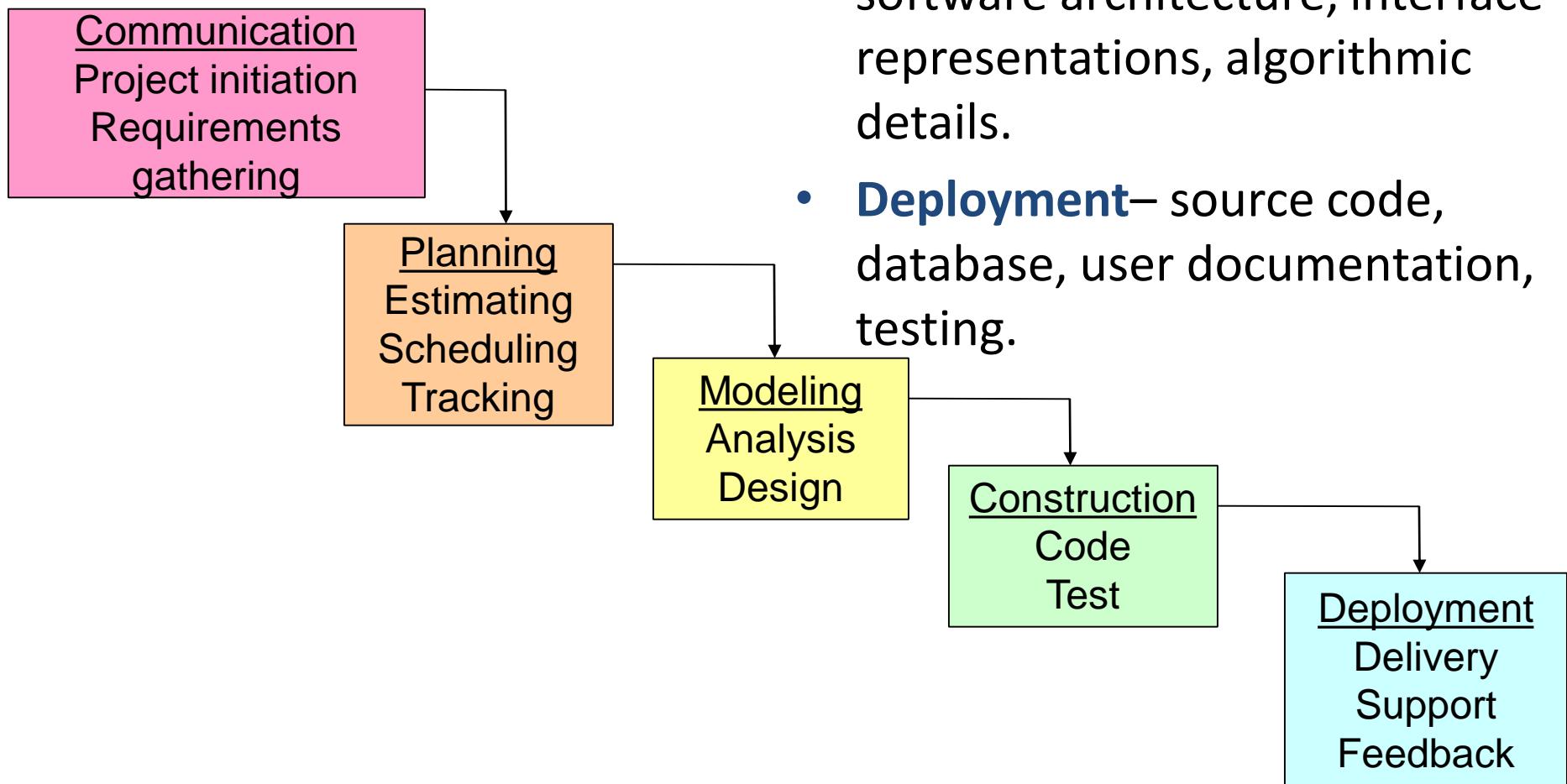
A **framework** that describes the activities performed at each **stage** of a software development project.

The System Development Life Cycle

Who participates in the system development life cycle?

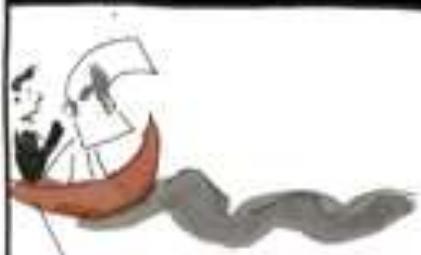


Waterfall Model



- **Requirements** – defines needed information, function, behavior, performance and interfaces.
- **Design** – data structures, software architecture, interface representations, algorithmic details.
- **Deployment** – source code, database, user documentation, testing.

THE NEW PRODUCT WATERFALL

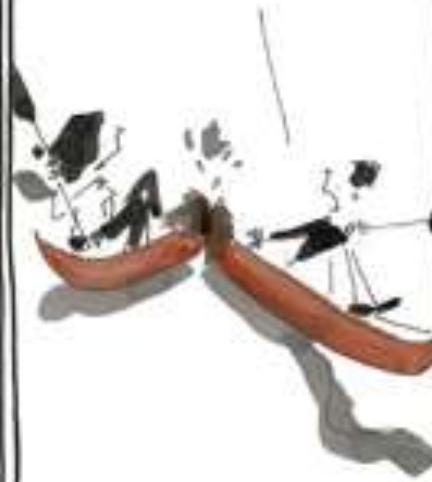


HOW DO WE
CHART OUR
ENTIRE COURSE
IF WE DON'T
KNOW WHAT'S
AHEAD?



WHATEVER
HAPPENS, JUST
KEEP PADDLING!

I WISH WE'D
DESIGNED FOR
THIS SCENARIO
UPFRONT



PATCH IT AS
BEST WE CAN.
NO TIME TO
CHANGE COURSE
NOW



PLAN

BUILD

TEST

LAUNCH

Waterfall Strengths

- Easy to understand, easy to use
- Provides structure to inexperienced staff
- Milestones are well understood
- Sets requirements stability
- Good for management control (plan, staff, track)
- Works well when quality is more important than cost or schedule

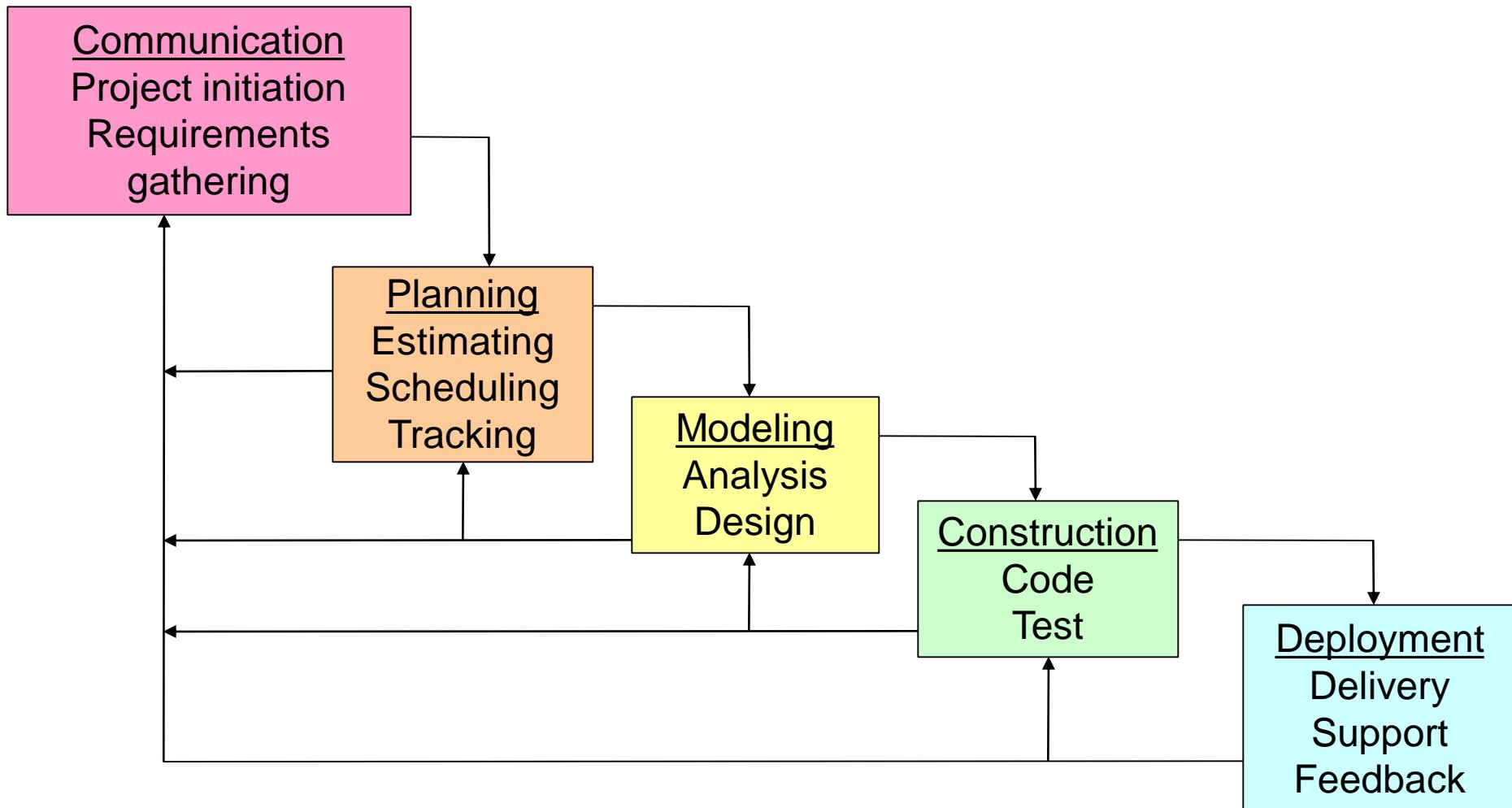
Waterfall Deficiencies

- All requirements must be known upfront
- Deliverables created for each phase are considered frozen – inhibits flexibility
- Can give a false impression of progress
- Does not reflect problem-solving nature of software development – iterations of phases
- Integration is one big bang at the end
- Little opportunity for customer to preview the system (until it may be too late)

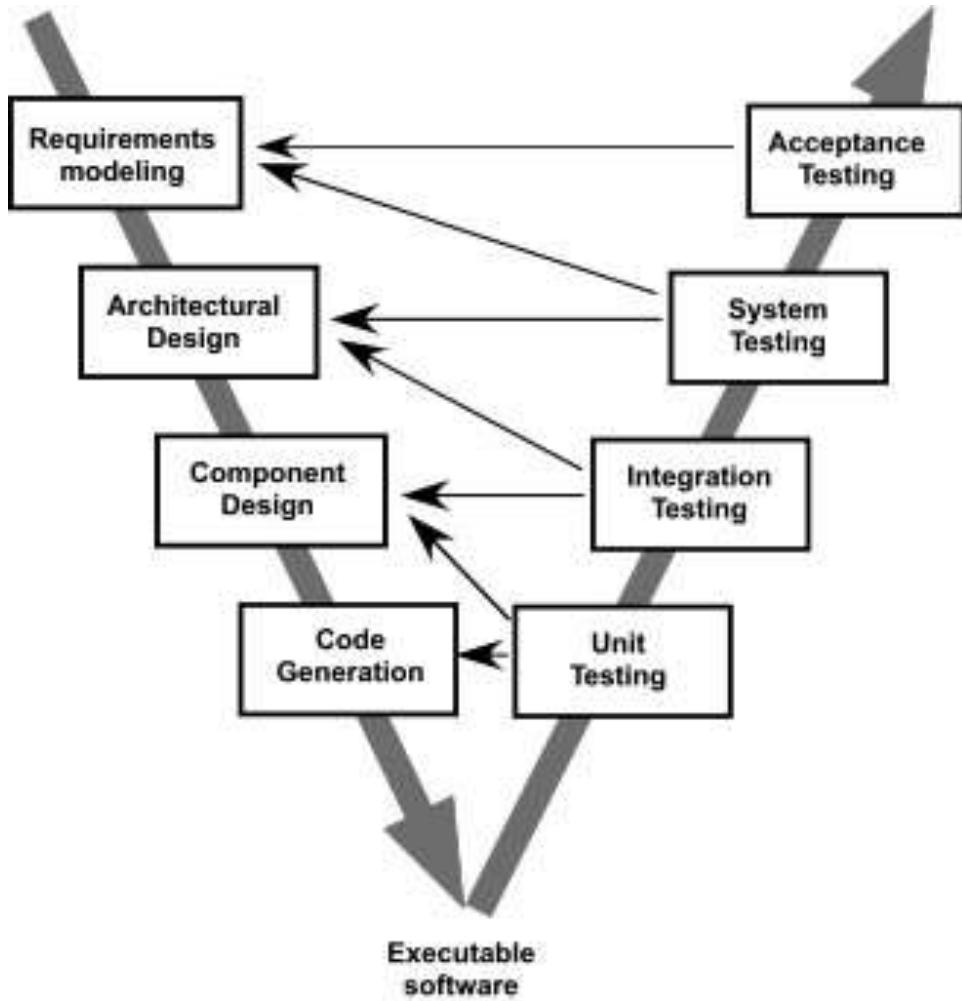
When to use the Waterfall Model

- Requirements are very well known
- Product definition is stable
- Technology is understood
- New version of an existing product
- Porting an existing product to a new platform.

Waterfall Model with Feedback (Diagram)



V-Shaped SDLC Model



- A variant of the Waterfall that emphasizes the **verification and validation** of the product.
- Testing of the product is planned in parallel with a corresponding phase of development

V-Shaped Steps

- Project and Requirements Planning – allocate resources
- Product Requirements and Specification Analysis – complete specification of the software system
- Architecture or High-Level Design – defines how software functions fulfill the design
- Detailed Design – develop algorithms for each architectural component
- Production, operation and maintenance – provide for enhancement and corrections
- System and acceptance testing – check the entire software system in its environment
- Integration and Testing – check that modules interconnect correctly
- Unit testing – check that each module acts as expected
- Coding – transform algorithms into software

V-Shaped Strengths

- Emphasize planning for verification and validation of the product in early stages of product development
- Each deliverable must be testable
- Project management can track progress by milestones
- Easy to use

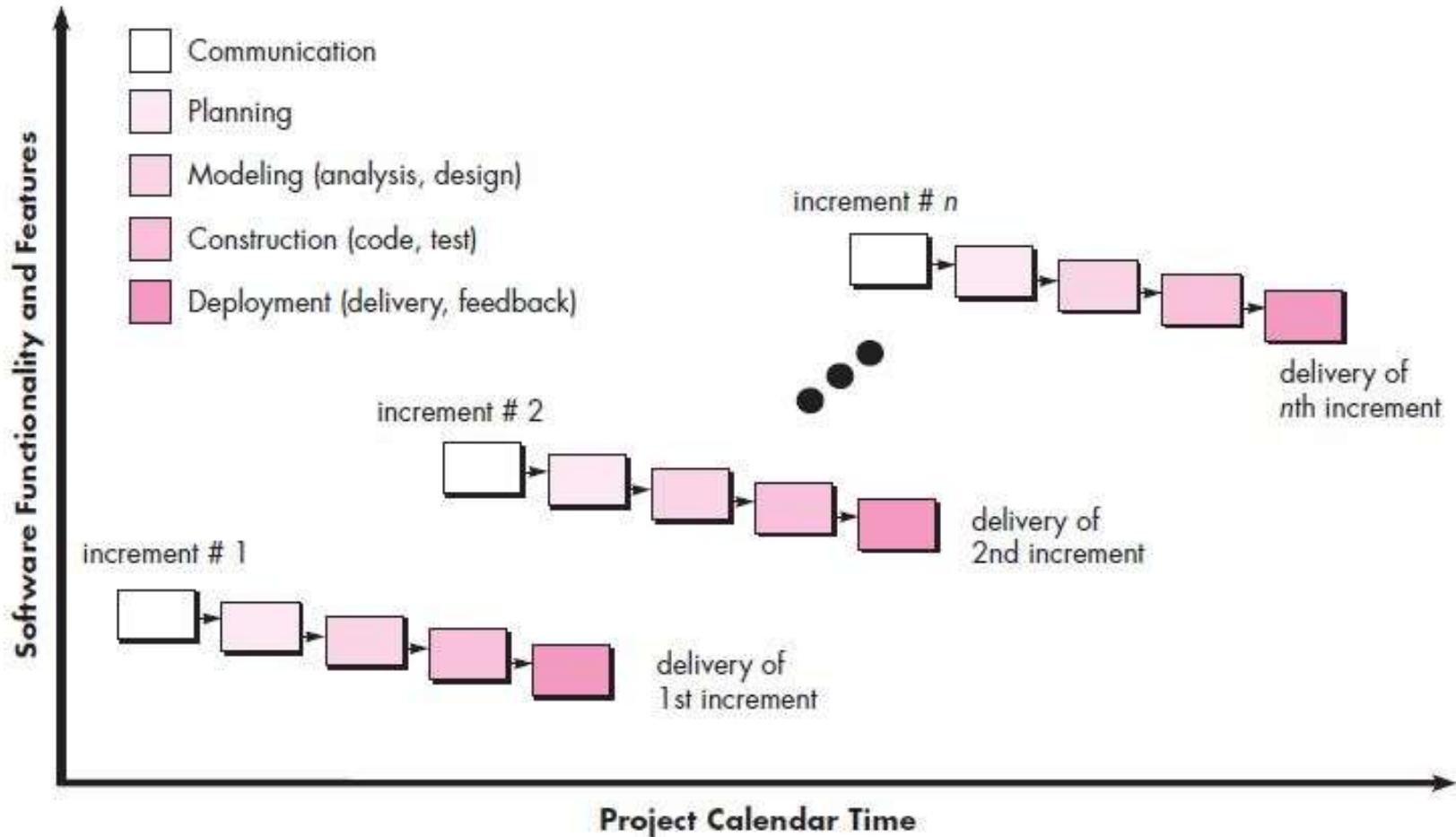
V-Shaped Weaknesses

- Does not easily handle concurrent events
- Does not handle iterations or phases
- Does not easily handle dynamic changes in requirements
- Does not contain risk analysis activities

When to use the V-Shaped Model

- Excellent choice for systems requiring high reliability – hospital patient control applications
- All requirements are known up-front
- When it can be modified to handle changing requirements beyond analysis phase
- Solution and technology are known

Incremental Model



Incremental Model

- Used when requirements are well understood
- Multiple independent deliveries are identified
- Work flow is in a linear (i.e., sequential) fashion within an increment and is staggered between increments
- Iterative in nature; focuses on an operational product with each increment
- Provides a needed set of functionality sooner while delivering optional components later
- Useful also when staffing is too short for a full-scale development

Incremental Model Strengths

- Develop high-risk or major functions first
- Each release delivers an operational product
- Customer can respond to each build
- Uses “divide and conquer” breakdown of tasks
- Lowers initial delivery cost
- Initial product delivery is faster
- Customers get important functionality early
- Risk of changing requirements is reduced

Incremental Model Weaknesses

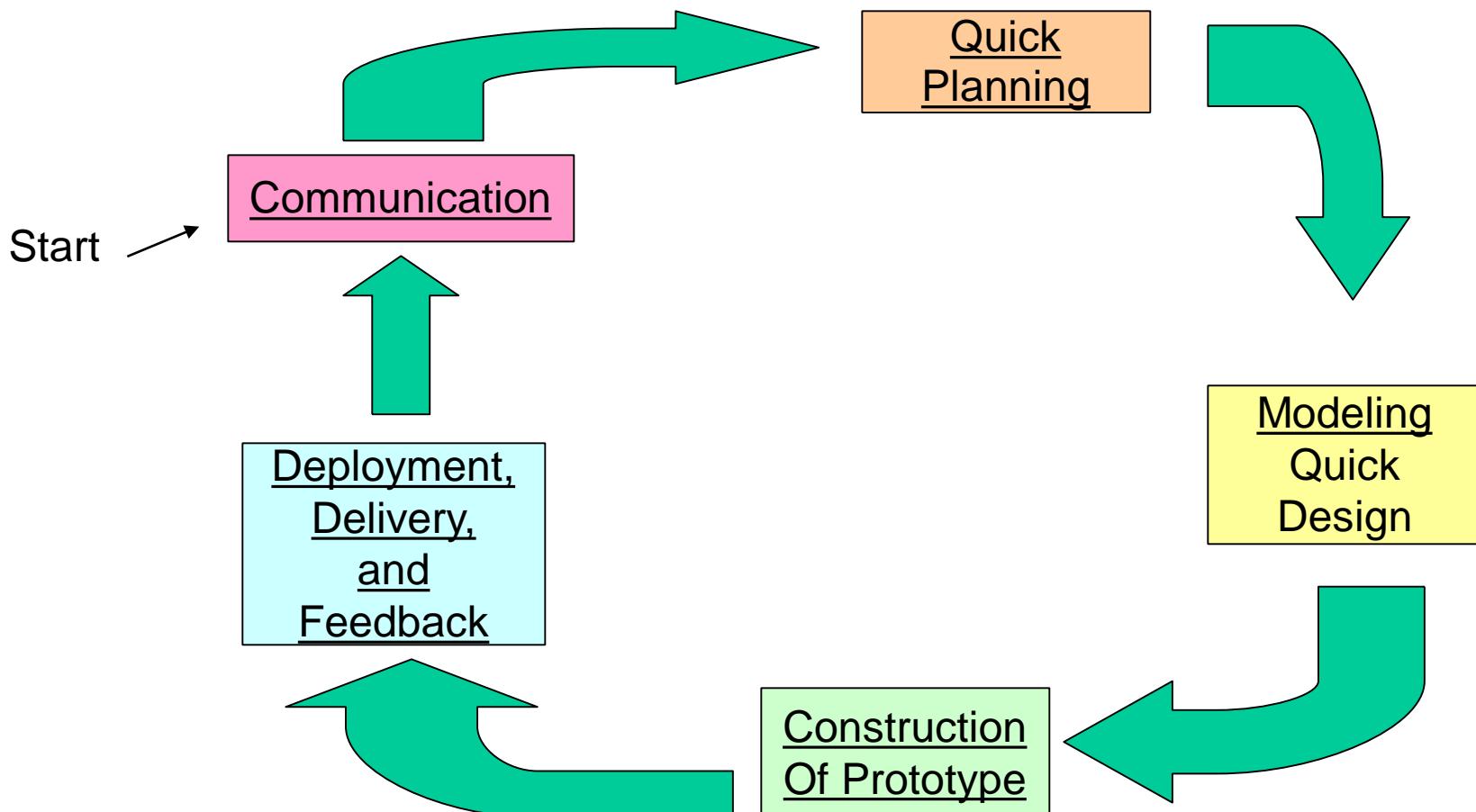
- Requires good planning and design
- Requires early definition of a complete and fully functional system to allow for the definition of increments
- Well-defined module interfaces are required (some will be developed long before others)
- Total cost of the complete system is not lower

When to use the Incremental Model

- Risk, funding, schedule, program complexity, or need for **early realization of benefits**.
- Most of the requirements are known up-front but are expected to **evolve over time**
- A need to **get basic functionality to the market early**
- On projects which have **lengthy development schedules**
- On a project with **new technology**

Evolutionary Process Model -

1. Prototyping



Evolutionary Prototyping Model

- Developers build a prototype during the requirements phase
- Prototype is evaluated by end users
- Users give corrective feedback
- Developers further refine the prototype
- When the user is satisfied, the prototype code is brought up to the standards needed for a final product.

Evolutionary Prototyping Steps

- A preliminary project plan is developed
- An partial high-level paper model is created
- The model is source for a partial requirements specification
- A prototype is built with basic and critical attributes
- The designer builds
 - the database
 - user interface
 - algorithmic functions
- The designer demonstrates the prototype, the user evaluates for problems and suggests improvements.
- This loop continues until the user is satisfied

Evolutionary Prototyping Strengths

- Customers can “see” the system requirements as they are being gathered
- Developers learn from customers
- A more accurate end product
- Unexpected requirements accommodated
- Allows for flexible design and development
- Steady, visible signs of progress produced
- Interaction with the prototype stimulates awareness of additional needed functionality

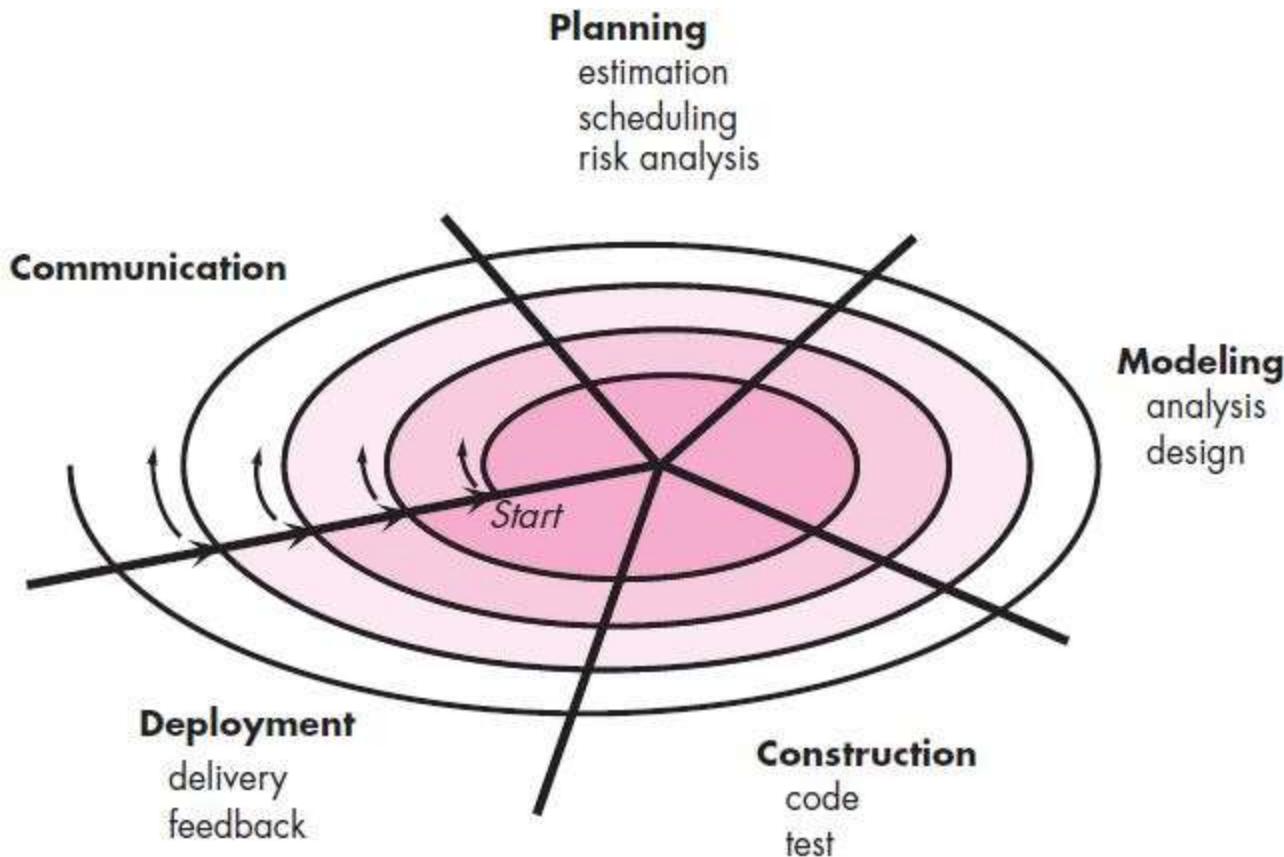
Evolutionary Prototyping Weaknesses

- Tendency to abandon structured program development for “code-and-fix” development
- Bad reputation for “quick-and-dirty” methods
- Overall maintainability may be overlooked
- The customer may want the prototype delivered.
- Process may continue forever (scope creep)

When to use Evolutionary Prototyping

- Requirements are unstable or have to be clarified
- As the requirements clarification stage of a waterfall model
- Develop user interfaces
- Short-lived demonstrations
- New, original development
- With the analysis and design portions of object-oriented development.

Evolutionary process – 2. Spiral Model



Spiral Model

- Invented by Dr. Barry Boehm in 1988 while working at TRW
- Follows an evolutionary approach
- Used when requirements are not well understood and risks are high
- Inner spirals focus on identifying software requirements and project risks; may also incorporate prototyping
- Outer spirals take on a classical waterfall approach after requirements have been defined, but permit iterative growth of the software
- Operates as a risk-driven model...a go/no-go decision occurs after each complete spiral in order to react to risk determinations
- Requires considerable expertise in risk assessment
- Serves as a realistic model for large-scale software development

Spiral Model Strengths

- Provides early indication of insurmountable risks, without much cost
- Users see the system early because of rapid prototyping tools
- Critical high-risk functions are developed first
- The design does not have to be perfect
- Users can be closely tied to all lifecycle steps
- Early and frequent feedback from users
- Cumulative costs assessed frequently

Spiral Model Weaknesses

- Time spent for evaluating risks too large for small or low-risk projects
- Time spent planning, resetting objectives, doing risk analysis and prototyping may be excessive
- The model is **complex**
- Risk assessment expertise is required
- Spiral may continue **indefinitely**
- Developers must be reassigned during non-development phase activities
- May be hard to define objective, verifiable milestones that indicate readiness to proceed through the next iteration

When to use Spiral Model

- When creation of a prototype is appropriate
- When costs and risk evaluation is important
- For medium to high-risk projects
- Long-term project commitment unwise because of potential changes to economic priorities
- Users are unsure of their needs
- Requirements are complex
- New product line
- Significant changes are expected (research and exploration)

Role Playing Game for SE's

- <http://www.youtube.com/watch?v=kkkl3LucxTY&feature=related>

General Weaknesses of Evolutionary Process Models

- 1) Prototyping poses a problem to project planning because of the uncertain number of iterations required to construct the product
- 2) Evolutionary software processes do not establish the maximum speed of the evolution
 - If too fast, the process will fall into chaos
 - If too slow, productivity could be affected
- 3) Software processes should focus first on flexibility and extensibility, and second on high quality
 - We should prioritize the speed of the development over zero defects
 - Extending the development in order to reach higher quality could result in late delivery



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



Statistic:

45% of features
in requirements
never used

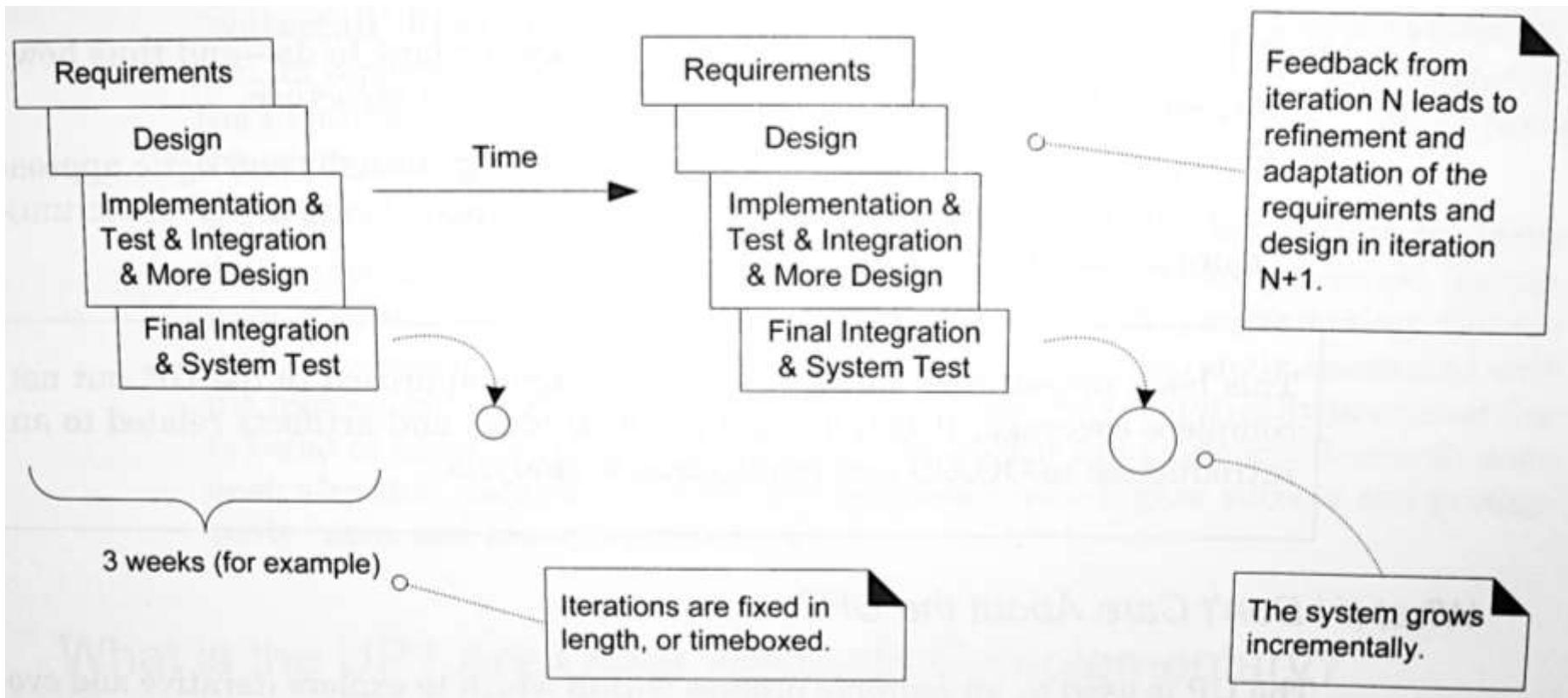


Statistic:

Early schedule
and estimates
off by up to 400%

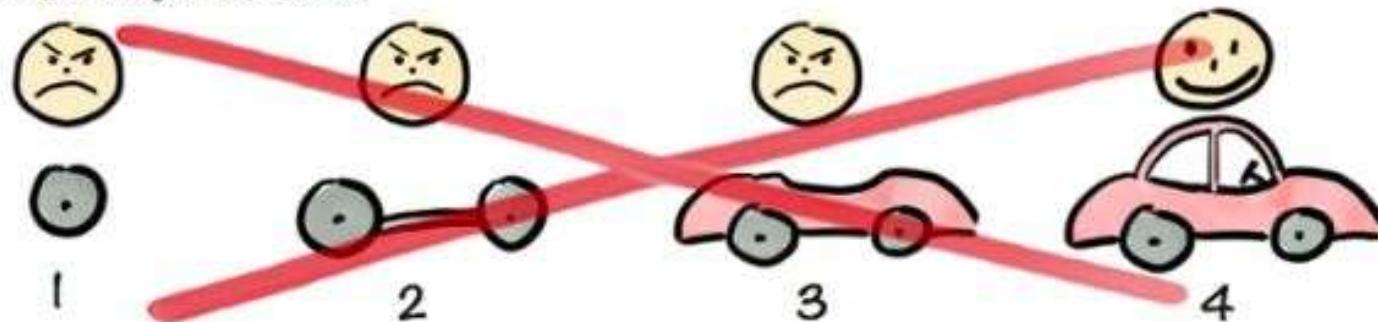
Iterative and incremental development

also called *iterative and evolutionary*

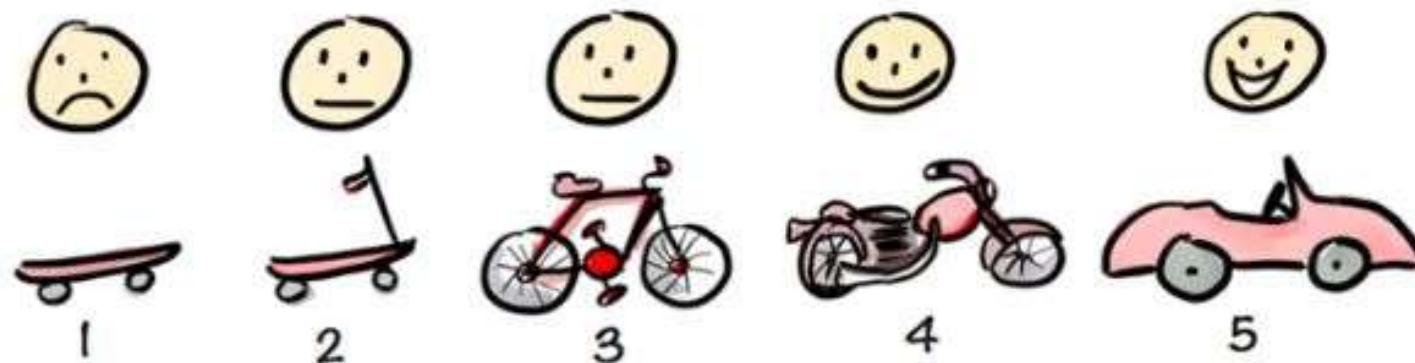


Iterative, Incremental Approach

Not like this....



Like this!



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

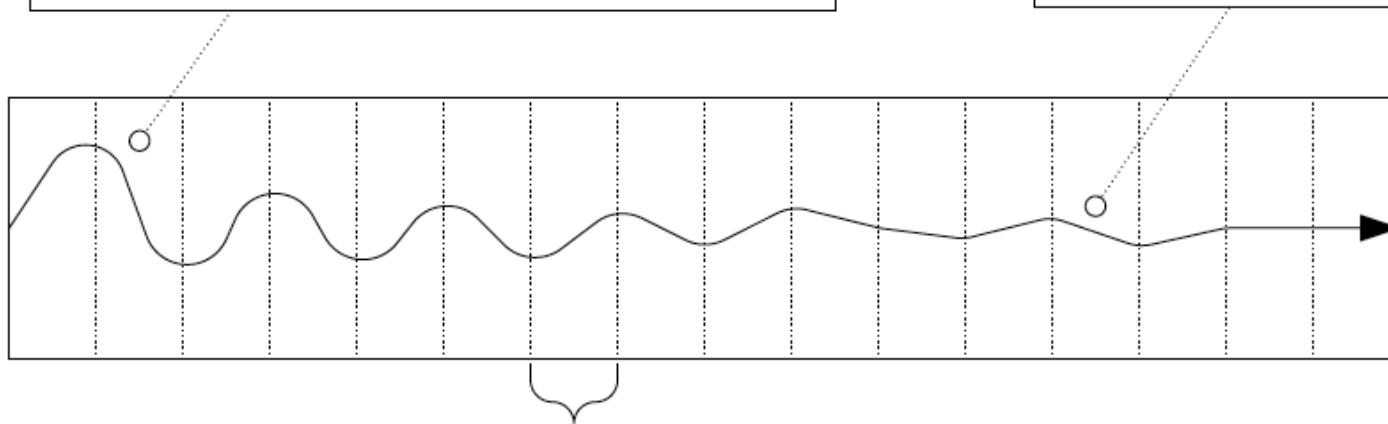


Yes, that's what I asked for, but now that I try it, what I really need is something slightly different.

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

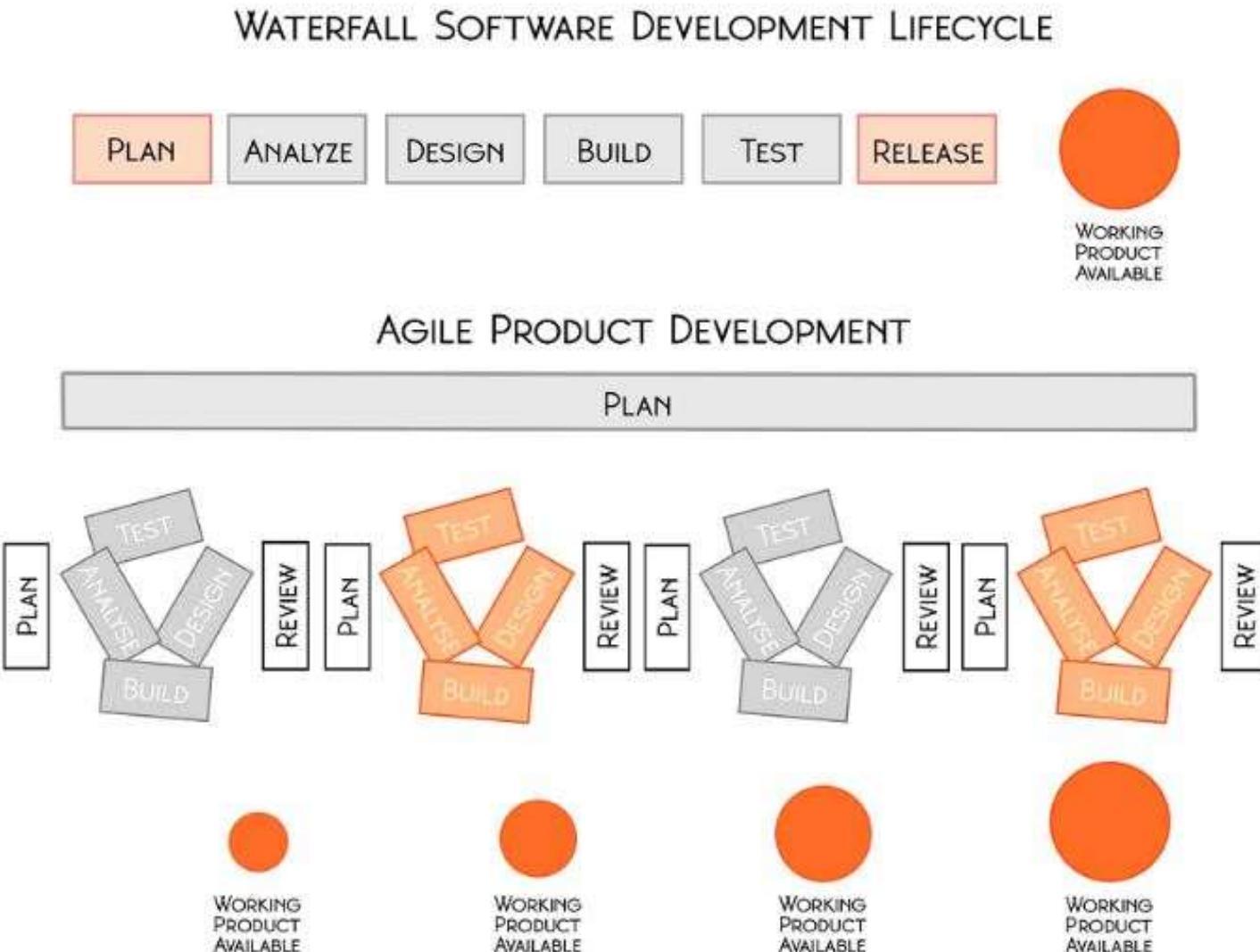
In particular, there are *agile methods*

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

Types of advice



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

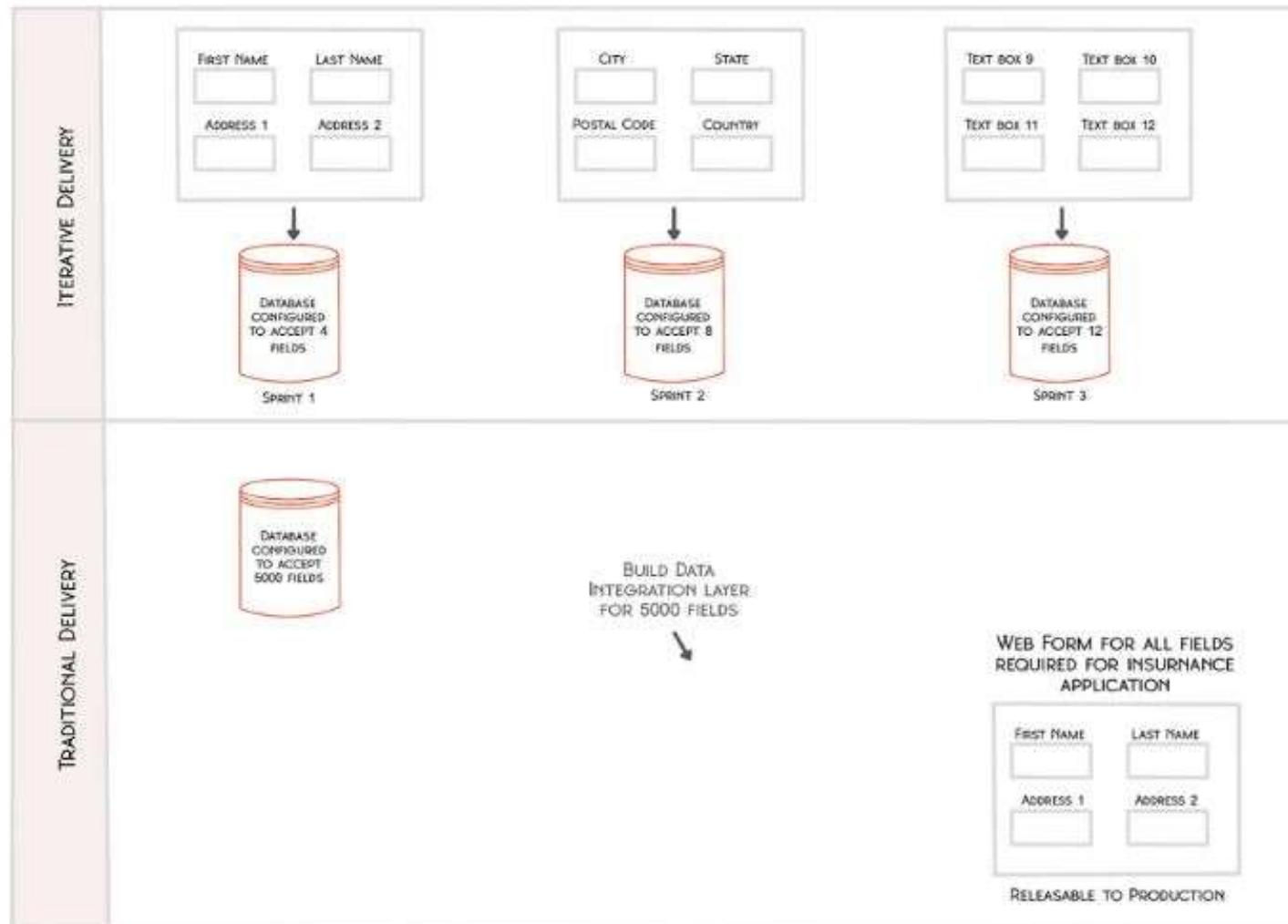
- Our plans and the ability to respond to changes is critical!
- Course of a project cannot be predicted far into the future.
 - Too many variables; not many good ways at estimating cost.
- Tempting to create a PERT or Gantt chart for whole project.
 - This does Not give novice managers control.
 - Can 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 facile.
- Agile process must be **adaptable** (12 principles)

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
 - ...



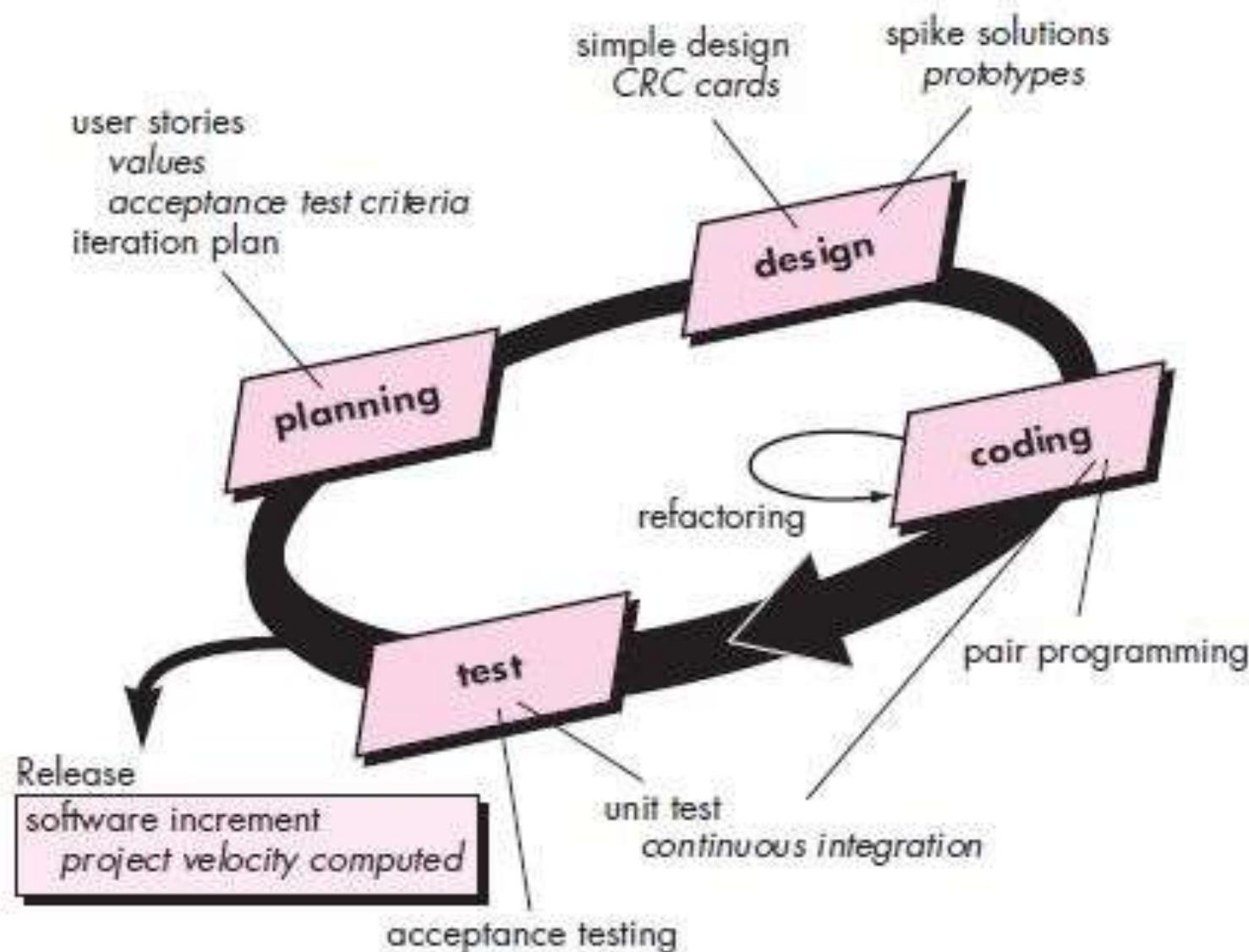
Extreme programming

- 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.

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 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

Prescribing medication

The record of the patient must be open for input. Click on the medication field and select either ‘current medication’, ‘new medication’ or ‘formulary’.

If you select ‘current medication’, you will be asked to check the dose; If you wish to change the dose, enter the new dose then confirm the prescription.

If you choose, ‘new medication’, the system assumes that you know which medication you wish to prescribe. Type the first few letters of the drug name. You will then see a list of possible drugs starting with these letters. Choose the required medication. You will then be asked to check that the medication you have selected is correct. Enter the dose then confirm the prescription.

If you choose ‘formulary’, you will be presented with a search box for the approved formulary. Search for the drug required then select it. You will then be asked to check that the medication you have selected is correct. Enter the dose then confirm the prescription.

In all cases, the system will check that the dose is within the approved range and will ask you to change it if it is outside the range of recommended doses.

After you have confirmed the prescription, it will be displayed for checking. Either click ‘OK’ or ‘Change’. If you click ‘OK’, your prescription will be recorded on the audit database. If you click ‘Change’, you reenter the ‘Prescribing medication’ process.

Examples of task cards for prescribing medication

Task 1: Change dose of prescribed drug

Task 2: Formulary selection

Task 3: Dose checking

Dose checking is a safety precaution to check that the doctor has not prescribed a dangerously small or large dose.

Using the formulary id for the generic drug name, lookup the formulary and retrieve the recommended maximum and minimum dose.

Check the prescribed dose against the minimum and maximum. If outside the range, issue an error message saying that the dose is too high or too low. If within the range, enable the 'Confirm' button.

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 a pair working together is more efficient than 2 programmers working separately.

XP References

Online references to XP at

- <http://www.extremeprogramming.org/>
- <http://c2.com/cgi/wiki?ExtremeProgrammingRoadmap>
- <http://www.xprogramming.com/>

A case study of SDLC in 21st century Healthcare

- Case Study Setting – The General Hospital
- Home Health and Study Overview – Home Health facility

SLDC in Action

- ✓ ***Problem Definition (Feasibility Study)***
- ✓ ***Requirements Analysis***
- ✓ ***Design***
- ✓ ***Implementation (Direct, Parallel, Single Location, Phased)***
- ✓ ***Maintenance/Support – Software Upgrades***

A must read for detailed study

[https://quod.lib.umich.edu/j/jsais/11880084.0001.103/--case-study-of-the-application-of-the-systems-development?rgn=main;view=fulltext#:~:text=In%20this%20classic%20representation%2C%20the,%2C%20Validation%20Test%3B%20and%207\)](https://quod.lib.umich.edu/j/jsais/11880084.0001.103/--case-study-of-the-application-of-the-systems-development?rgn=main;view=fulltext#:~:text=In%20this%20classic%20representation%2C%20the,%2C%20Validation%20Test%3B%20and%207))

Conclusions

- The professional goal of every software engineer, and every development team, is to deliver the highest possible value to our employers and customers.
 - And yet, our projects fail, or fail to deliver value, at a dismaying rate.
- Though well intentioned, the **upward spiral of process inflation** is culpable for at least some of this failure.
- The principles and values of agile software development were formed as a way
 - to help teams break the cycle of process inflation, and
 - to focus on simple techniques for reaching their goals.
- At the time of this writing there were many agile processes to choose from. These include
 - SCRUM,
 - Crystal,
 - Feature Driven Development (FDD),
 - Adaptive Software Development (ADP), and most significantly,
 - Extreme Programming (XP).
 - Others...

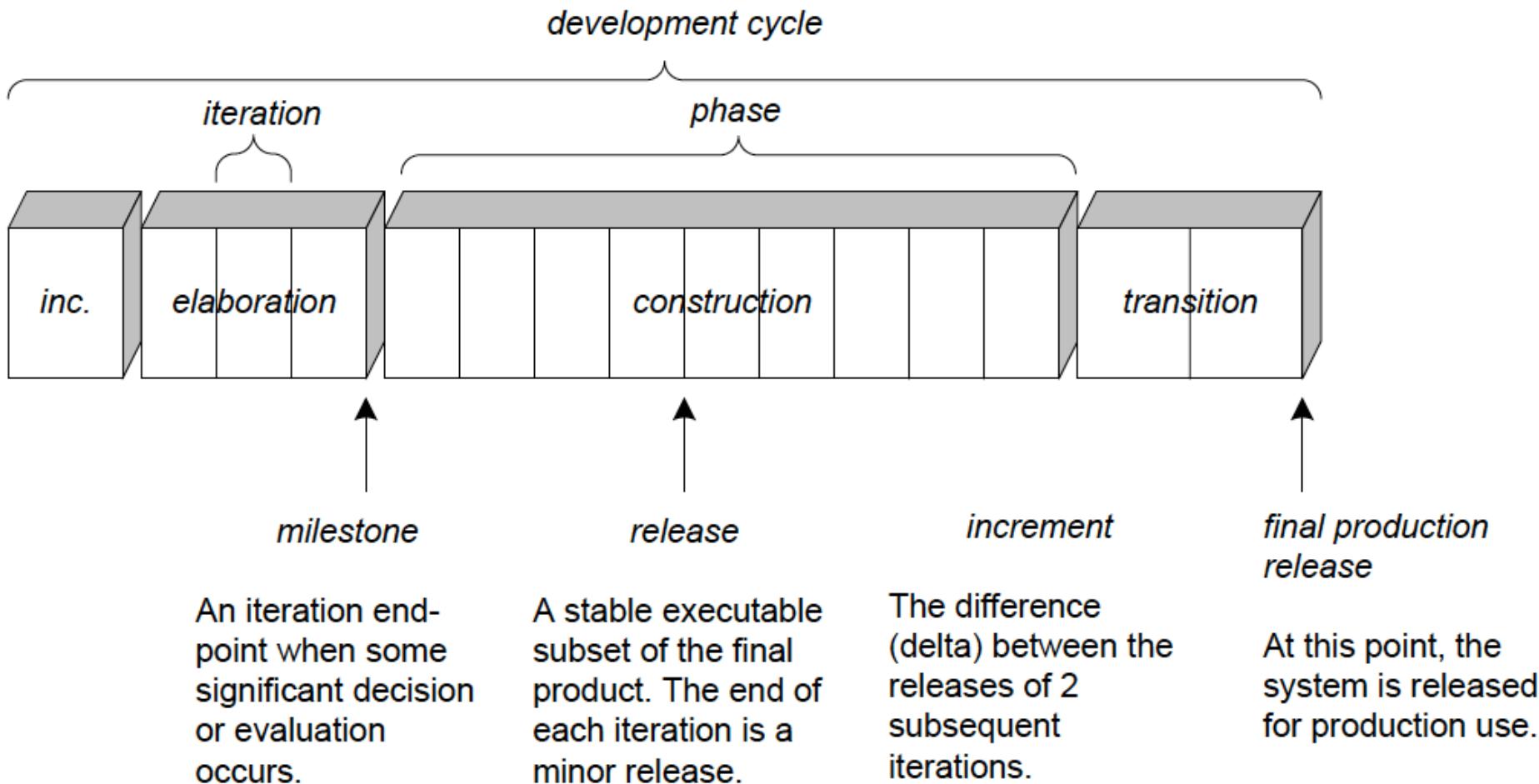
So what is this *UP* thing that Larman keeps talking about?

- UP = Unified Process
 - See also Rational Unified Process (RUP; IBM's refinement)
- Process *framework*
 - Customizable: other methods/practices can be plugged in
 - Agile methods for example!
 - But has some practices of its own as well
 - Iterative and incremental
 - Defines phases across series of iterations

Phases of UP

- **Inception:** Approximate vision, business case, scope, vague estimates
- **Elaboration:** Refined vision, iterative implementation of core architecture, resolution of high risks, most requirements, more realistic estimates
- **Construction:** Iterative implementation of remaining lower risk and easier elements, prep for deployment
- **Transition:** Beta tests, deployment

Phases of UP (cont'd)

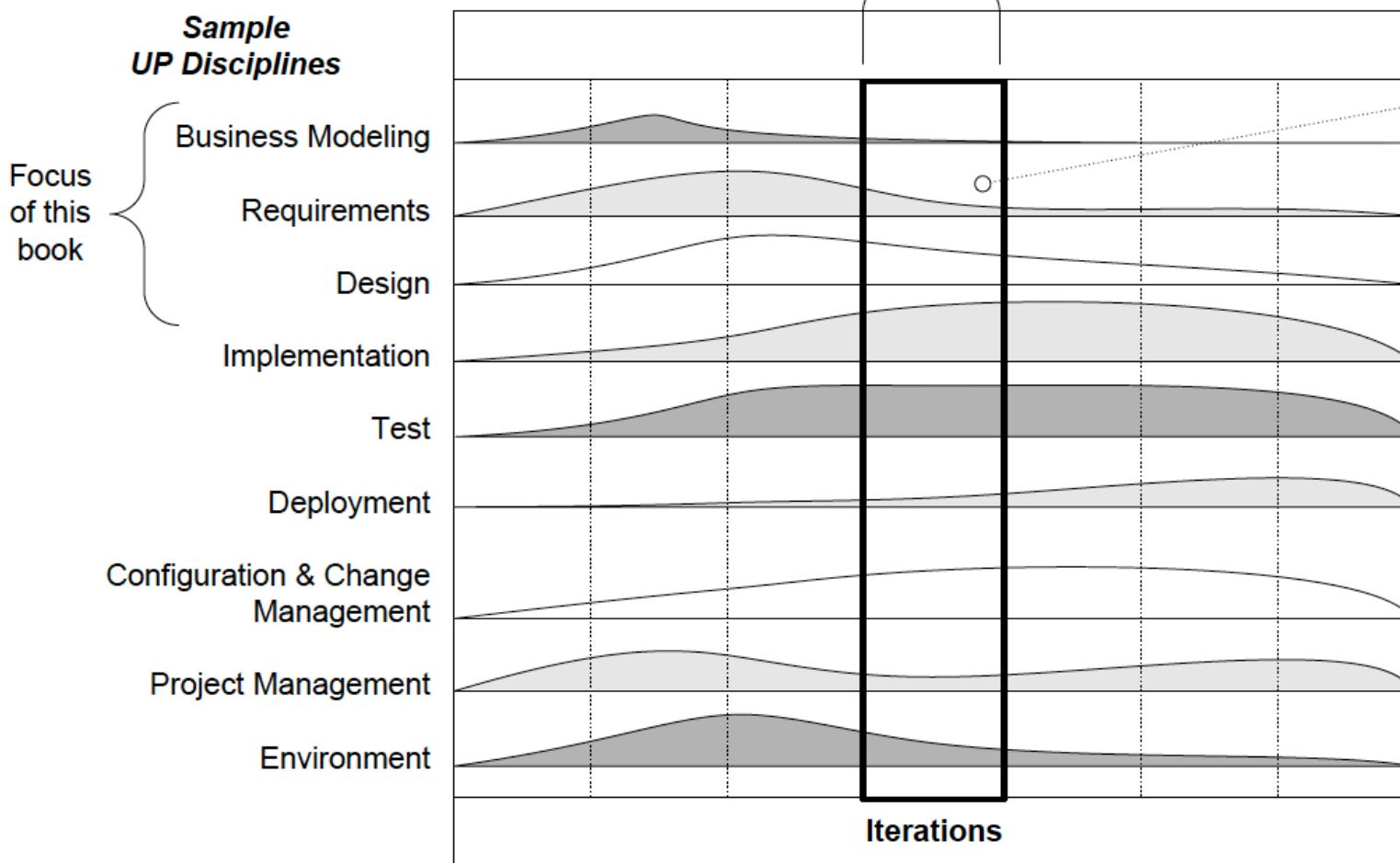


UP *Disciplines*

- Set of activities in one subject area
- Examples:
 - Business modeling
 - Requirements
 - Design
 - Test
- Each discipline typically associated with particular *artifacts* (e.g., code, models, documents)

Discipline activity across iterations

A four-week iteration (for example).
A mini-project that includes work in most disciplines, ending in a stable executable.



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

This example is suggestive, not literal.

Disciplines/artifacts across phases

Discipline	Artifact Iteration-*	Incep. 11	Elab. El. .En	Const. CL.Cn	Trans. T1..T2
Business Modeling	Domain Model		s		
Requirements	Use-Case Model	s	r		
	Vision	s	r		
	Supplementary Specification	s	r		
	Glossary	s	r		
Design	Design Model		s	r	
	SW Architecture Document		s		
	Data Model		s	r	
Implementation	Implementation Model		s	r	r
Project Management	SW Development Plan	s	r	r	r
Testing	Test Model		s	r	
Environment	Development Case	s	r		

s = started r = refined

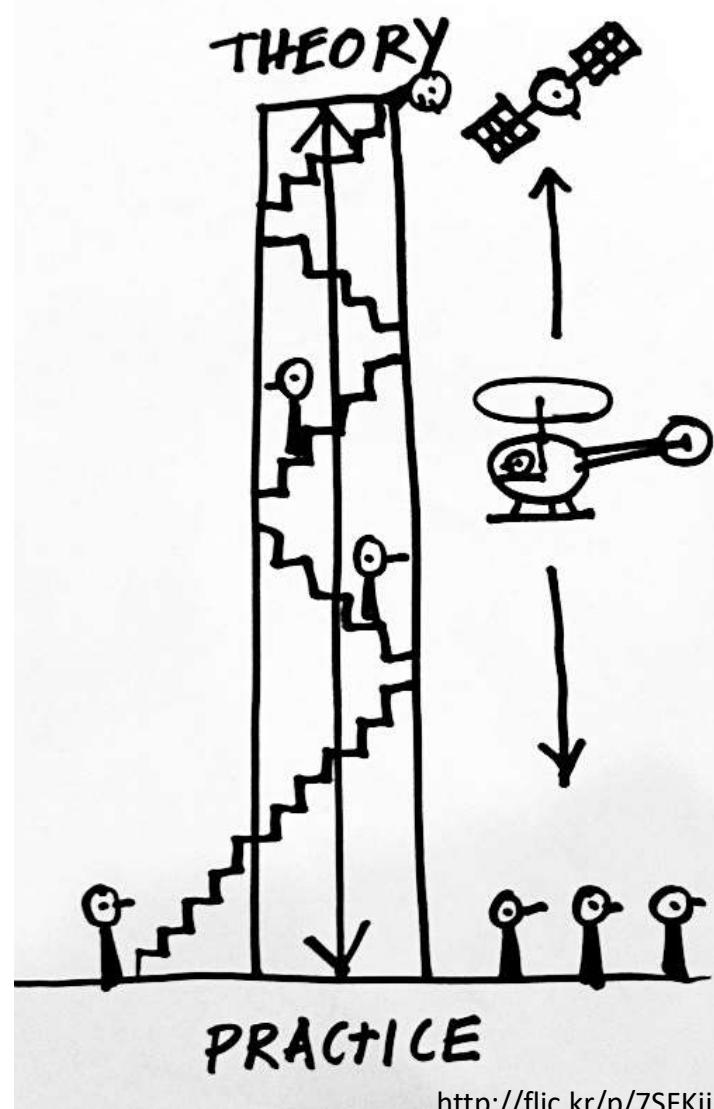
A word about *agile modeling*

(quoting Larman)

Experienced analysts and
modelers know the
secret of modeling:

The purpose of modeling
(sketching UML, ...) is
primarily to *understand*,
not to document.

Thus, we favor hand-drawn
diagrams over typeset ones



You know you're doing UP wrong when...



- Define most requirements before starting design or implementation
- Spend days/weeks modeling before programming
- Think inception=requirements, elaboration=design, construction=implementation
- Think elaboration is to fully define models
- Believe that iterations should be 3 months
- Think you need to create many formal documents
- Try to plan project in detail from start to finish

Rapid Application Model (RAD)

- Requirements planning phase (a workshop utilizing structured discussion of business problems)
- User description phase – automated tools capture information from users
- Construction phase – productivity tools, such as code generators, screen generators, etc. inside a time-box. (“Do until done”)
- Cutover phase -- installation of the system, user acceptance testing and user training

RAD Strengths

- Reduced cycle time and improved productivity with fewer people means lower costs
- Time-box approach mitigates cost and schedule risk
- Customer involved throughout the complete cycle minimizes risk of not achieving customer satisfaction and business needs
- Focus moves from documentation to code (WYSIWYG).
- Uses modeling concepts to capture information about business, data, and processes.

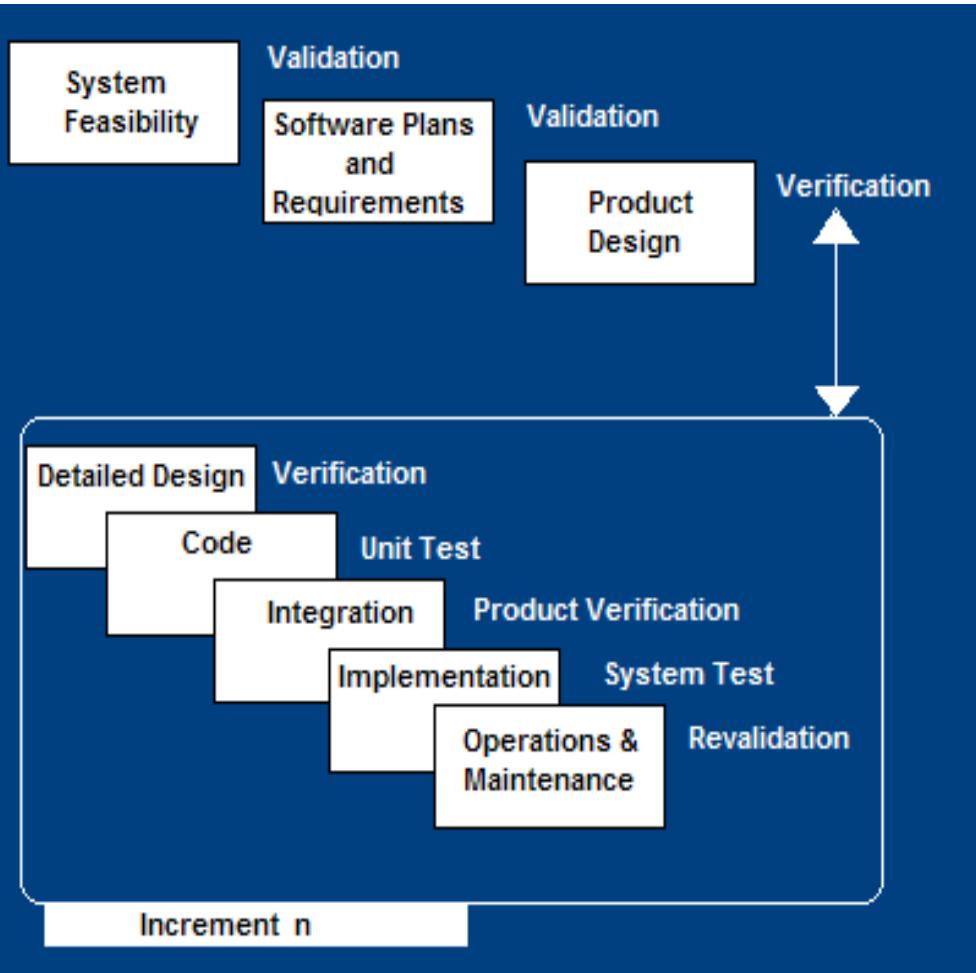
RAD Weaknesses

- Accelerated development process must give quick responses to the user
- Risk of never achieving closure
- Hard to use with legacy systems
- Requires a system that can be modularized
- Developers and customers must be committed to rapid-fire activities in an abbreviated time frame.

When to use RAD

- Reasonably well-known requirements
- User involved throughout the life cycle
- Project can be time-boxed
- Functionality delivered in increments
- High performance not required
- Low technical risks
- System can be modularized

Incremental SDLC Model



- Construct a partial implementation of a total system
- Then slowly add increased functionality
- The incremental model prioritizes requirements of the system and then implements them in groups.
- Each subsequent release of the system adds function to the previous release, until all designed functionality has been implemented.

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

Some Agile Methods

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

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 Practices (1-6)

1. **Planning game** – determine scope of the next release by combining business priorities and technical estimates
2. **Small releases** – put a simple system into production, then release new versions in very short cycle
3. **Metaphor** – all development is guided by a simple shared story of how the whole system works
4. **Simple design** – system is designed as simply as possible (extra complexity removed as soon as found)
5. **Testing** – programmers continuously write unit tests; customers write tests for features
6. **Refactoring** – programmers continuously restructure the system without changing its behavior to remove duplication and simplify

XP Practices (7 – 12)

7. **Pair-programming** -- all production code is written with two programmers at one machine
8. **Collective ownership** – anyone can change any code anywhere in the system at any time.
9. **Continuous integration** – integrate and build the system many times a day – every time a task is completed.
10. **40-hour week** – work no more than 40 hours a week as a rule
11. **On-site customer** – a user is on the team and available full-time to answer questions
12. **Coding standards** – programmers write all code in accordance with rules emphasizing communication through the code

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)

Feature Driven Design (FDD)

Five FDD process activities

1. Develop an overall model – Produce class and sequence diagrams from chief architect meeting with domain experts and developers.
2. Build a features list – Identify all the features that support requirements. The features are functionally decomposed into Business Activities steps within Subject Areas.
Features are functions that can be developed in two weeks and expressed in client terms with the template: <action> <result> <object>
i.e. Calculate the total of a sale
3. Plan by feature -- the development staff plans the development sequence of features
4. Design by feature -- the team produces sequence diagrams for the selected features
5. Build by feature – the team writes and tests the code

<http://www.nebulon.com/articles/index.html>

Dynamic Systems Development Method (DSDM)

Applies a framework for RAD and short time frames

Paradigm is the 80/20 rule

- majority of the requirements can be delivered in a relatively short amount of time.

DSDM Principles

1. Active user involvement imperative (Ambassador users)
2. DSDM teams empowered to make decisions
3. Focus on frequent product delivery
4. Product acceptance is fitness for business purpose
5. Iterative and incremental development - to converge on a solution
6. Requirements initially agreed at a high level
7. All changes made during development are reversible
8. Testing is integrated throughout the life cycle
9. Collaborative and co-operative approach among all stakeholders essential

DSDM Lifecycle

- Feasibility study
- Business study – prioritized requirements
- Functional model iteration
 - risk analysis
 - Time-box plan
- Design and build iteration
- Implementation

Adaptive SDLC

Combines RAD with software engineering best practices

- Project initiation
- Adaptive cycle planning
- Concurrent component engineering
- Quality review
- Final QA and release

Adaptive Steps

1. Project initialization – determine intent of project
2. Determine the project time-box (estimation duration of the project)
3. Determine the optimal number of cycles and the time-box for each
4. Write an objective statement for each cycle
5. Assign primary components to each cycle
6. Develop a project task list
7. Review the success of a cycle
8. Plan the next cycle

Tailored SDLC Models

- Any one model does not fit all projects
- If there is nothing that fits a particular project, pick a model that comes close and modify it for your needs.
- Project should consider risk but complete spiral too much – start with spiral & pare it done
- Project delivered in increments but there are serious reliability issues – combine incremental model with the V-shaped model
- Each team must pick or customize a SDLC model to fit its project

Agile Web references

DePaul web site has links to many Agile references

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

Quality – the degree to which the software satisfies stated and implied requirements

- Absence of system crashes
- Correspondence between the software and the users' expectations
- Performance to specified requirements

Quality must be controlled because it lowers production speed, increases maintenance costs and can adversely affect business

Quality Assurance Plan

- The plan for quality assurance activities should be in writing
- Decide if a separate group should perform the quality assurance activities
- Some elements that should be considered by the plan are: defect tracking, unit testing, source-code tracking, technical reviews, integration testing and system testing.

Quality Assurance Plan

- **Defect tracing** – keeps track of each defect found, its source, when it was detected, when it was resolved, how it was resolved, etc
- **Unit testing** – each individual module is tested
- **Source code tracing** – step through source code line by line
- **Technical reviews** – completed work is reviewed by peers
- **Integration testing** -- exercise new code in combination with code that already has been integrated
- **System testing** – execution of the software for the purpose of finding defects.



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



Statistic:

45% of features
in requirements
never used

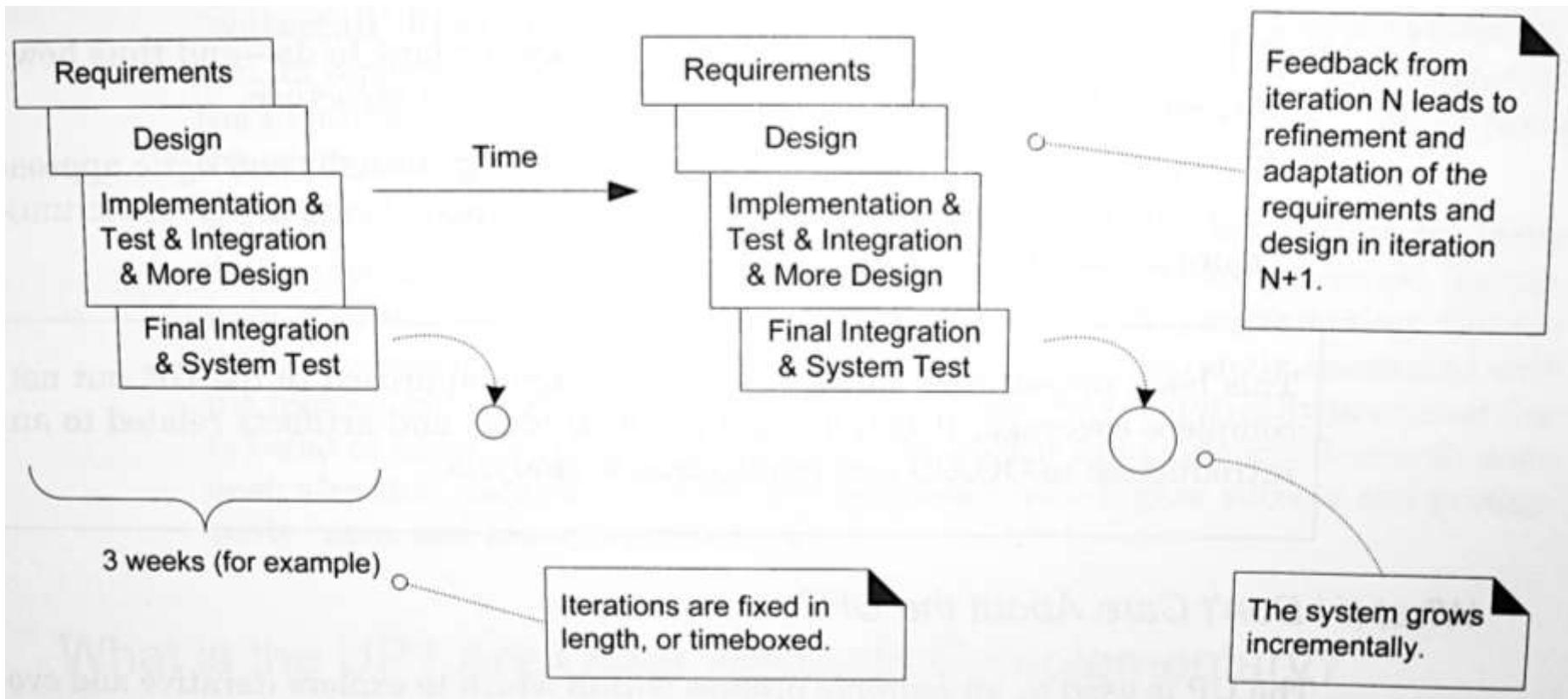


Statistic:

Early schedule
and estimates
off by up to 400%

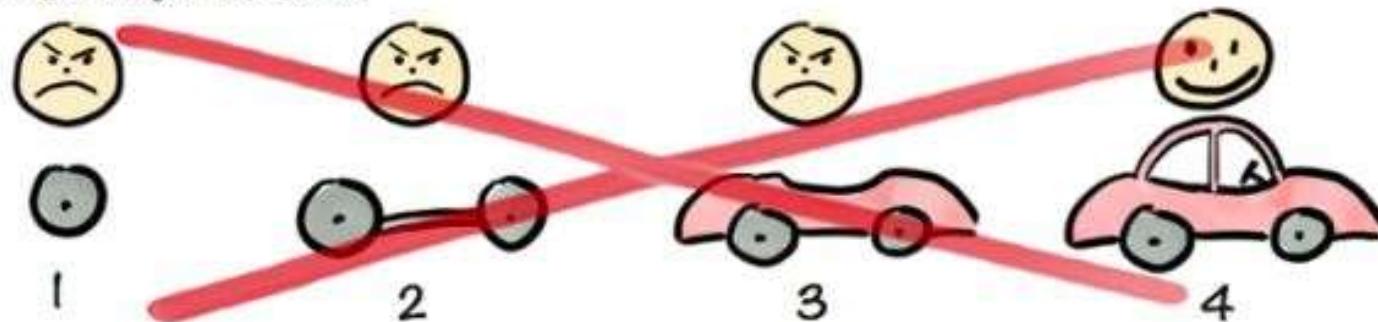
Iterative and incremental development

also called *iterative and evolutionary*

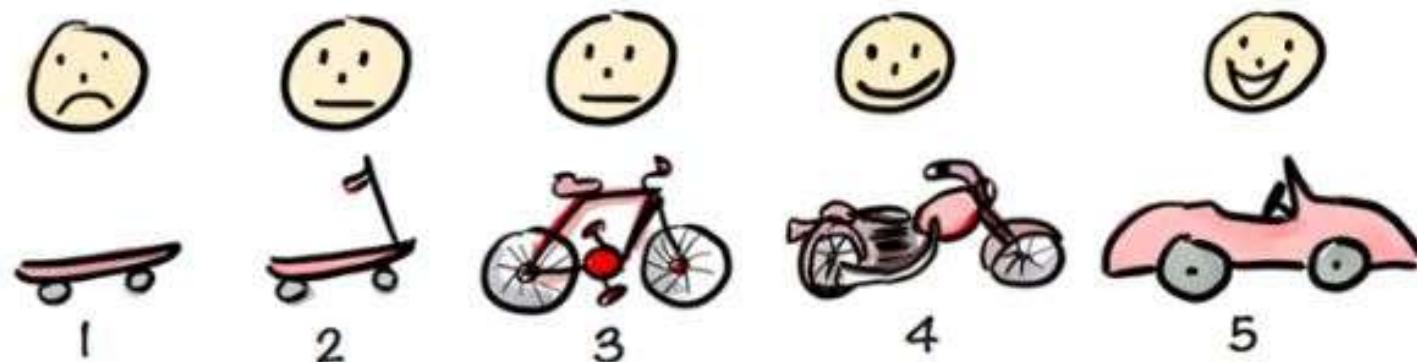


Iterative, Incremental Approach

Not like this....



Like this!



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

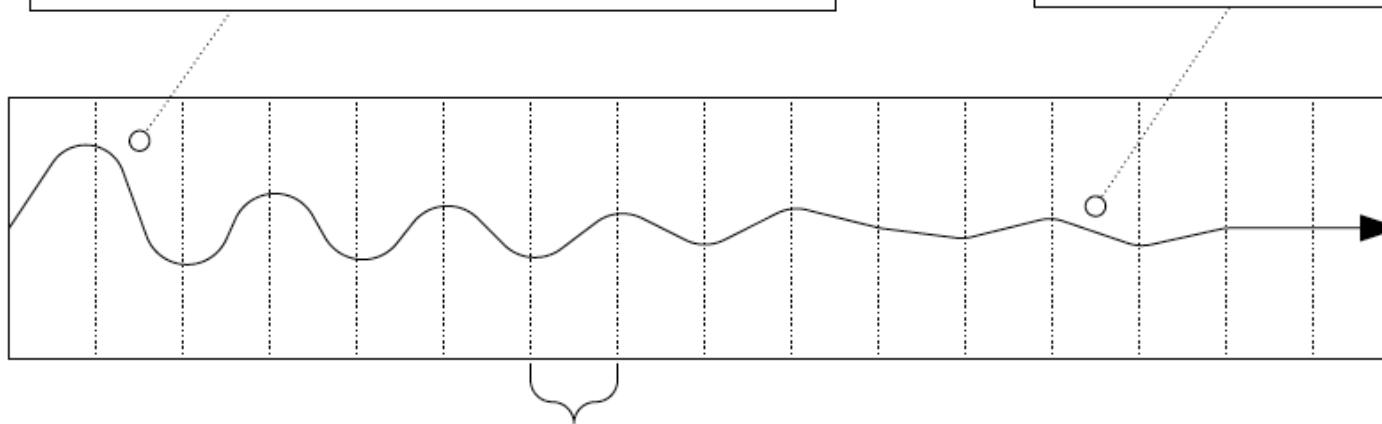


Yes, that's what I asked for, but now that I try it, what I really need is something slightly different.

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

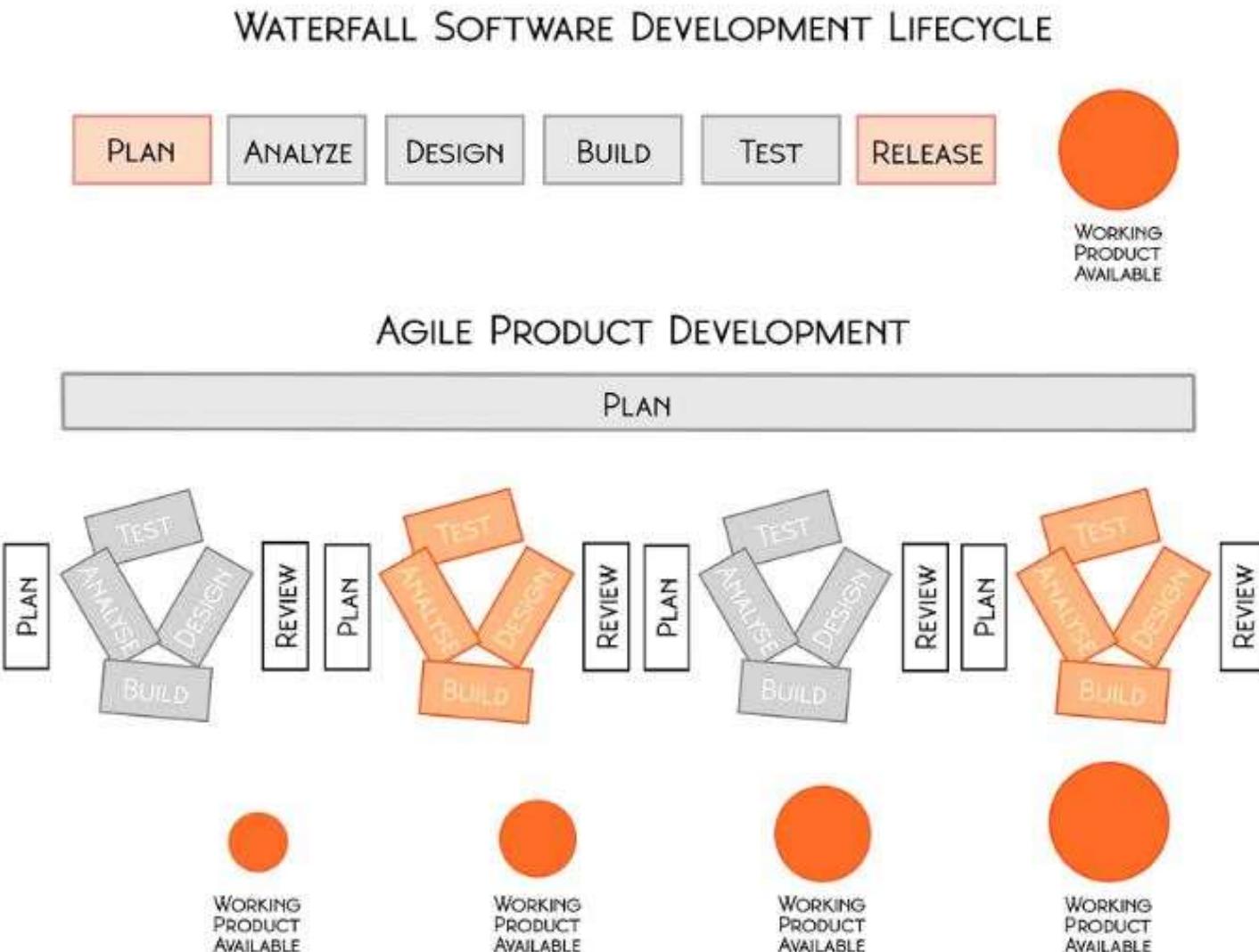
In particular, there are *agile methods*

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

Types of advice



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

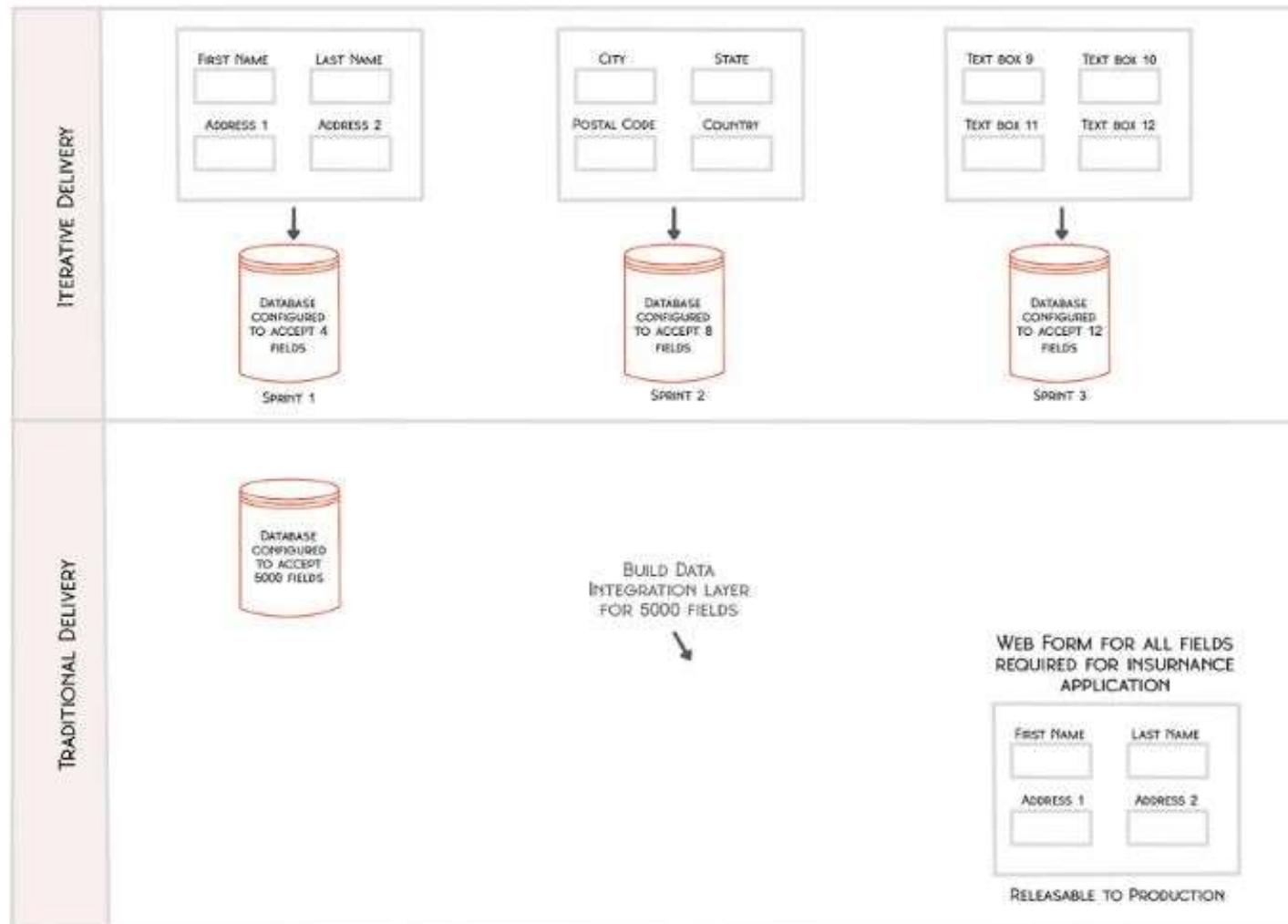
- Our plans and the ability to respond to changes is critical!
- Course of a project cannot be predicted far into the future.
 - Too many variables; not many good ways at estimating cost.
- Tempting to create a PERT or Gantt chart for whole project.
 - This does Not give novice managers control.
 - Can 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 facile.
- Agile process must be **adaptable** (12 principles)

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

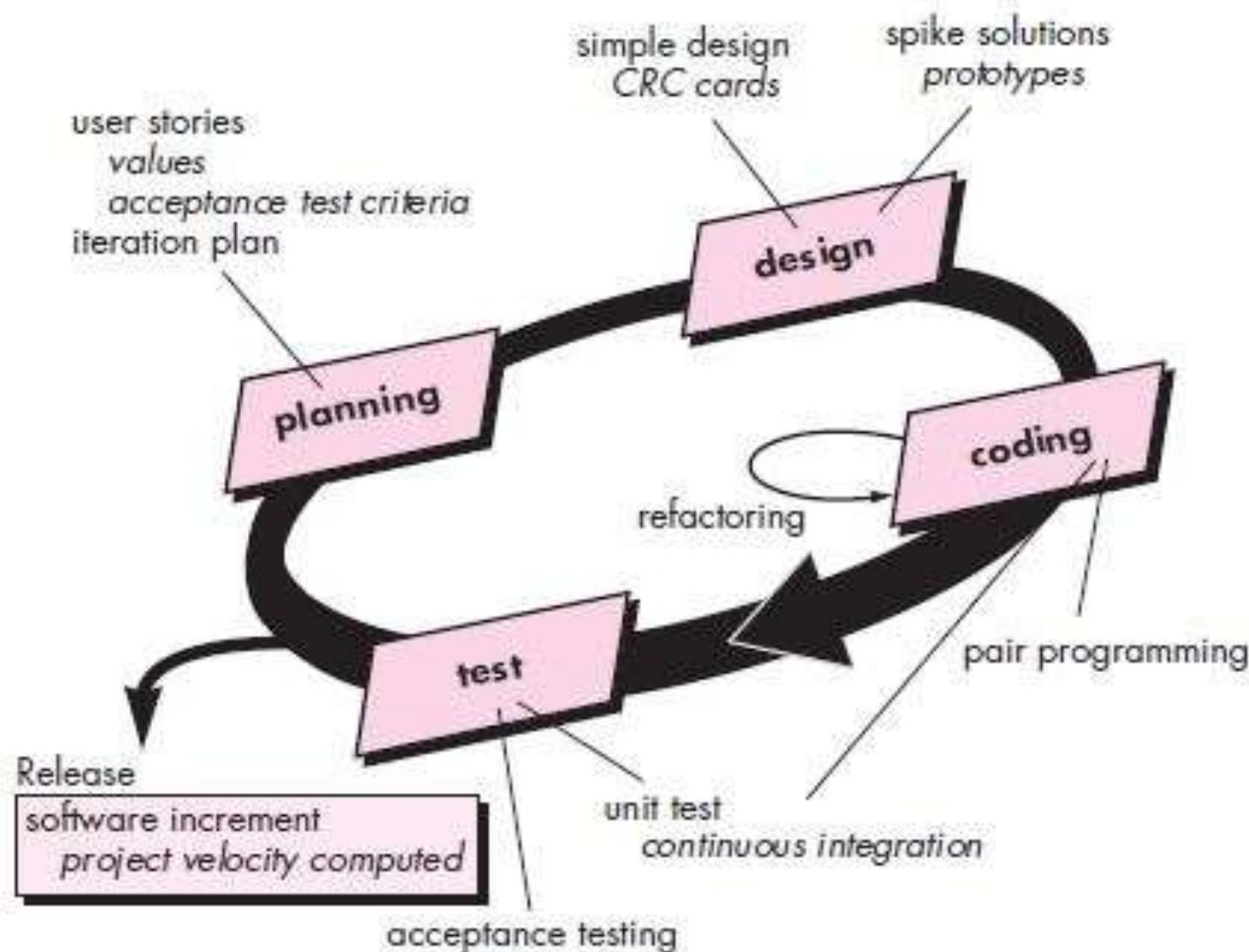
Coding is the key activity throughout a software project

- 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

Prescribing medication

The record of the patient must be open for input. Click on the medication field and select either ‘current medication’, ‘new medication’ or ‘formulary’.

If you select ‘current medication’, you will be asked to check the dose; If you wish to change the dose, enter the new dose then confirm the prescription.

If you choose, ‘new medication’, the system assumes that you know which medication you wish to prescribe. Type the first few letters of the drug name. You will then see a list of possible drugs starting with these letters. Choose the required medication. You will then be asked to check that the medication you have selected is correct. Enter the dose then confirm the prescription.

If you choose ‘formulary’, you will be presented with a search box for the approved formulary. Search for the drug required then select it. You will then be asked to check that the medication you have selected is correct. Enter the dose then confirm the prescription.

In all cases, the system will check that the dose is within the approved range and will ask you to change it if it is outside the range of recommended doses.

After you have confirmed the prescription, it will be displayed for checking. Either click ‘OK’ or ‘Change’. If you click ‘OK’, your prescription will be recorded on the audit database. If you click ‘Change’, you reenter the ‘Prescribing medication’ process.

Examples of task cards for prescribing medication

Task 1: Change dose of prescribed drug

Task 2: Formulary selection

Task 3: Dose checking

Dose checking is a safety precaution to check that the doctor has not prescribed a dangerously small or large dose.

Using the formulary id for the generic drug name, lookup the formulary and retrieve the recommended maximum and minimum dose.

Check the prescribed dose against the minimum and maximum. If outside the range, issue an error message saying that the dose is too high or too low. If within the range, enable the 'Confirm' button.

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 a pair working together is more efficient than 2 programmers working separately.

XP References

Online references to XP at

- <http://www.extremeprogramming.org/>
- <http://c2.com/cgi/wiki?ExtremeProgrammingRoadmap>
- <http://www.xprogramming.com/>

Scrum

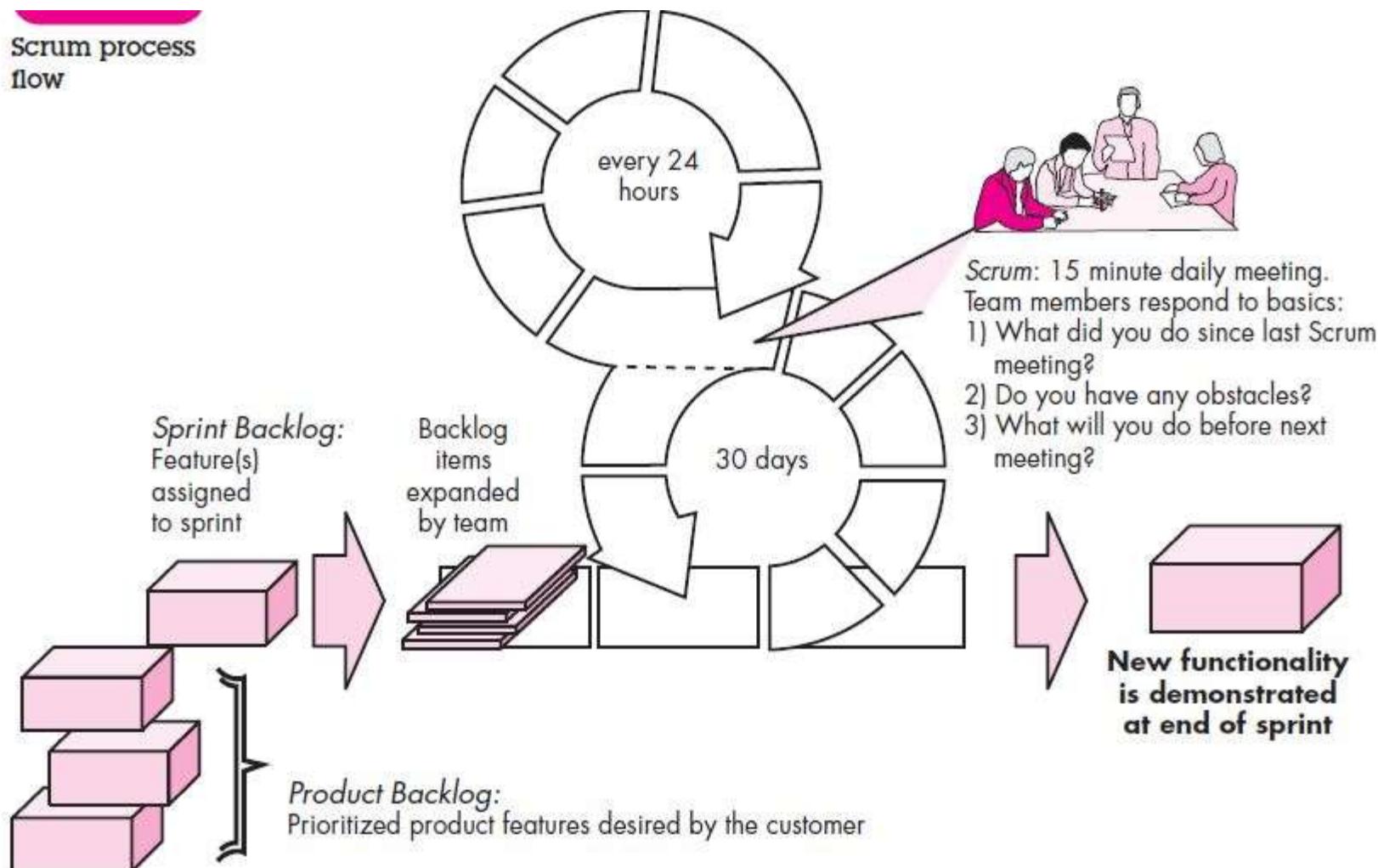
- ✧ Scrum (the name is derived from an activity that occurs during a rugby match) - Jeff Sutherland 1990s.
- ✧ Project Manager's job: - Deliver needed system on time within budget
- ✧ The Scrum approach - manage the iterations
- ✧ There are three phases in Scrum.
 - outline planning phase - general picture and architecture
 - Sprint cycles releasing increments of the system.
 - The project closure phase - final delivery, documentation and review of lessons learned.

Phases in Scrum

- Requirements
- Analysis
- Design
- Evolution and
- Delivery

Scrum Process Flow

Scrum process flow



Components of Scrum

- requirements, analysis, design, evolution, and delivery
- emphasizes the use of a set of software process patterns –

Backlog,

Sprints (30 days),

Scrum meetings and

Demos

Eg. Shopping Cart development

Priority	Backlog Task	Number of Days to Complete	Sprint Number
1	As a user, I need to be able to create a user account	7	1
2	As a user, I need to be able to edit my user account profile	7	1
3	As a user, I need to be able to place items in my cart	5	2
4	As a user, I need to be able to view items in my cart	5	2
5	I need to be able to remove items from my cart	4	2
6	I need to be able to ask questions about items in my cart	7	3
7	I need to be able to pay for items in my cart	7	3

The Sprint cycle

- ✧ Every 2–4 weeks (a fixed length).
- ✧ 1) **Project team with customer:** Look at product backlog - select stories to implement
- ✧ 2) implement with all customer communication through **scrum master** (protecting pgmr at this point)
 - ✧ Scrum master has project manager role during sprint
 - ✧ Daily 15 min meetings
 - ✧ Stand up often
 - ✧ Team presents progress and impediments
 - ✧ Scrum master tasked with removing impediments
- ✧ 3) Review system release with user

Scrum benefits

- ✧ The product is broken down into a set of manageable and understandable chunks.
- ✧ Unstable requirements do not hold up progress.
- ✧ The whole team have **visibility of everything** and consequently team communication is improved.
- ✧ Customers see **on-time delivery** of increments and gain feedback on how the product works.
- ✧ Trust between customers and developers is established and a positive culture is created in which everyone expects the project to succeed.

Summary

- ✧ Plan Driven (Ex: Waterfall) vs Incremental (Ex: Agile)
 - ✧ Structure and benefits and downfalls
- ✧ XP - an implementation of Agile - **Power to the Programmer**
 - ✧ User story requirements
 - ✧ Test driven design with continual retest and integration
 - ✧ Pair Programming
 - ✧ Refactoring encouraged
- ✧ Scrum - project management of Agile using sprints
 - ✧ Iterations of full team contact / Scrum master protection of programmers / full team release review

A case study of SDLC in 21st century Healthcare

Case Study Setting – The General Hospital

A must read for detailed study

[https://quod.lib.umich.edu/j/jsais/11880084.0001.103/--case-study-of-the-application-of-the-systems-development?rgn=main;view=fulltext#:~:text=In%20this%20classic%20representation%2C%20the,%2C%20Validation%20Test%3B%20and%207\)](https://quod.lib.umich.edu/j/jsais/11880084.0001.103/--case-study-of-the-application-of-the-systems-development?rgn=main;view=fulltext#:~:text=In%20this%20classic%20representation%2C%20the,%2C%20Validation%20Test%3B%20and%207))

Home Health and Study Overview

- Home Health, or Home Care, is the portion of health care that is carried out at the patient's home or residence. It is a participatory arrangement that eliminates the need for constant trips to the hospital for routine procedures. For example, patients take their own blood pressure (or heart rate, glucose level, etc.) using a device hooked up near their bed at home. The results are transmitted to the hospital (or in this case, the Home Health facility near General Hospital) electronically and are immediately processed, inspected, and monitored by attending staff.
- In addition, there is a Lifeline feature available to elderly or other homebound individuals. The unit includes a button worn on a necklace or bracelet that the patient can push should they need assistance (“Home Health”, 2010). Periodically, clinicians (e.g., nurses, physical therapists, etc.) will visit the patient in their home to monitor their progress and perform routine inspections and maintenance on the technology.

Home Health and Study Overview

- The author was approached by his neighbor, a retired accounting faculty member who is a volunteer at General Hospital. He had been asked by hospital administration to investigate the acquisition, and eventual purchase, of software to facilitate and help coordinate the Home Health care portion of their business. After an initial meeting to offer help and familiarize ourselves with the task at hand, we met with staff (i.e., both management and the end-users) at the Home Health facility to begin our research.

A case study of SDLC in 21st century Healthcare

SLDC in Action

- ✓ ***Problem Definition (Feasibility Study*** – Technical, Economical and Operational)
- ✓ ***Requirements Analysis***
- ✓ ***Design***
- ✓ ***Implementation (Direct, Parallel, Single Location, Phased)***
- ✓ ***Maintenance/Support – Software Upgrades***

A must read for detailed study

[https://quod.lib.umich.edu/j/jsais/11880084.0001.103/--case-study-of-the-application-of-the-systems-development?rgn=main;view=fulltext#:~:text=In%20this%20classic%20representation%2C%20the,%2C%20Validation%20Test%3B%20and%207\)](https://quod.lib.umich.edu/j/jsais/11880084.0001.103/--case-study-of-the-application-of-the-systems-development?rgn=main;view=fulltext#:~:text=In%20this%20classic%20representation%2C%20the,%2C%20Validation%20Test%3B%20and%207))

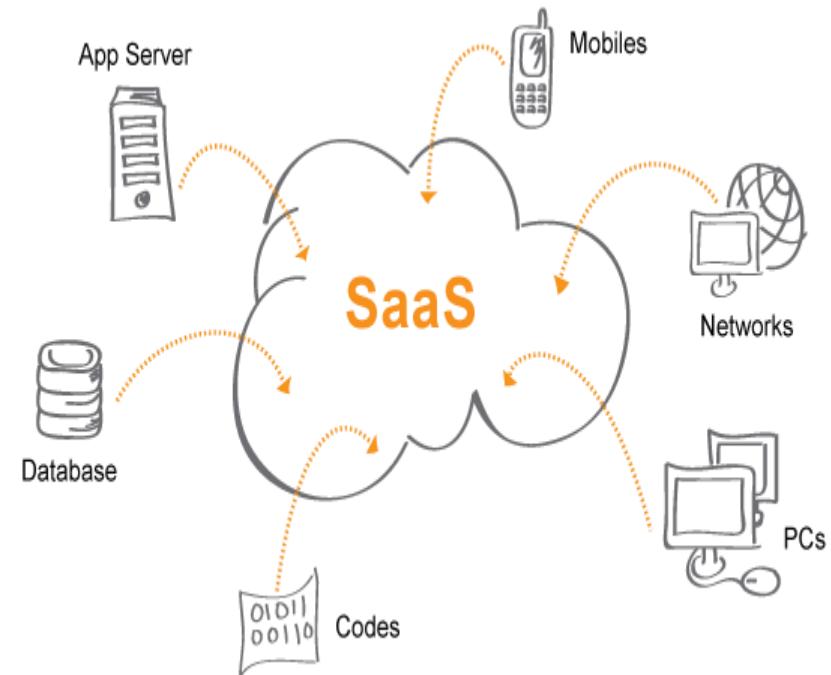
Self Study

Software as a Service (SaaS)

- • Definition: Software as a Service (SaaS), a.k.a. on-demand software, is a software delivery model in which software and its associated data are hosted centrally and accessed using a thin-client, usually a web browser over the internet. – Wikipedia
- • Simply put, SaaS is a method for delivering software that provides remote access to software as a webbased service. The software service can be purchased with a monthly fee and pay as you go.

Software as a Service (SaaS)

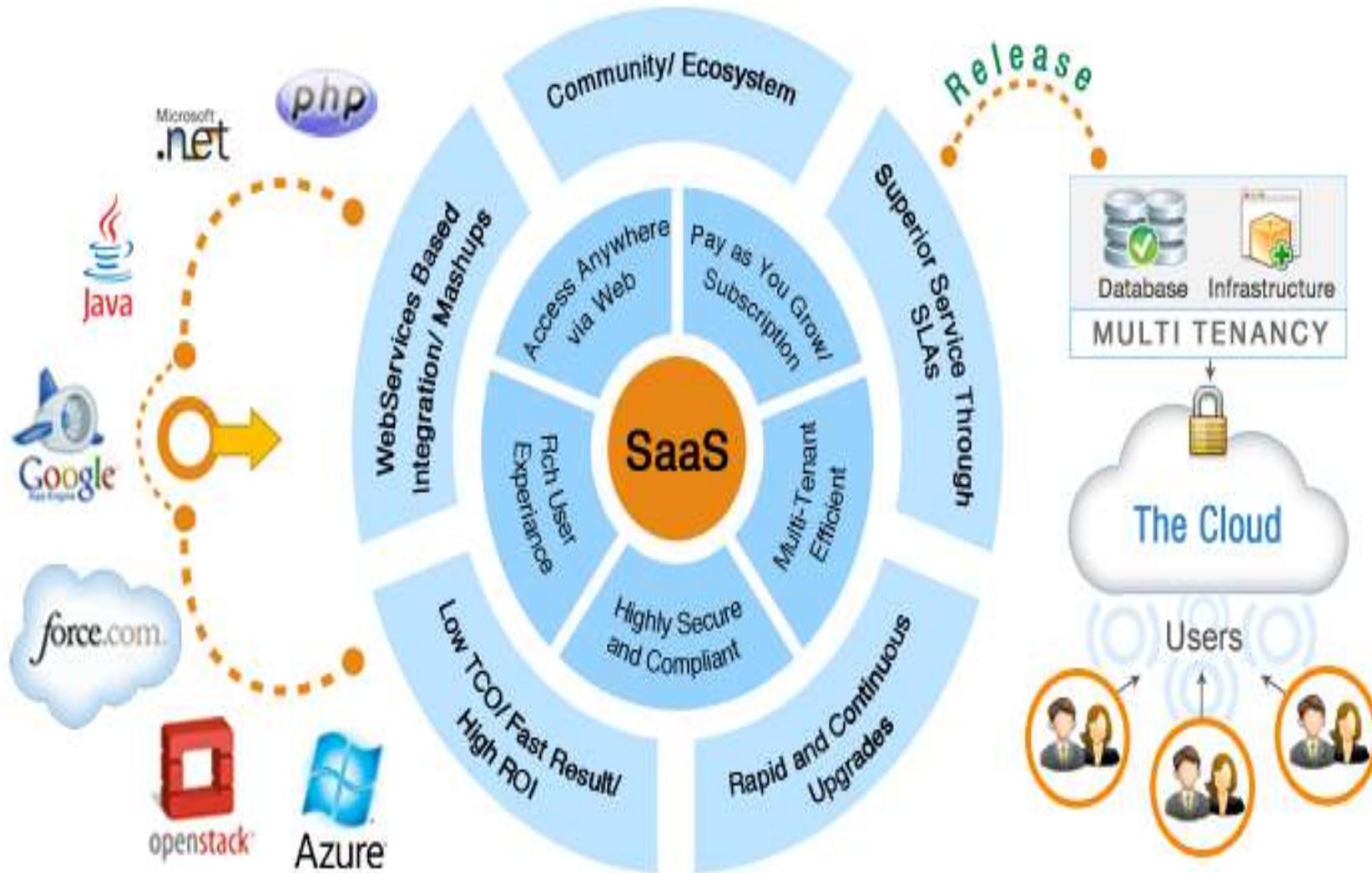
- SaaS is a model of software deployment where an application is hosted as a service provided to customers across the Internet.
- SaaS alleviates the burden of software maintenance/support
 - but users relinquish control over software versions and requirements.
- Terms that are used in this sphere include
 - **Platform as a Service (PaaS)** and
 - **Infrastructure as a Service (IaaS)**



Applicability of SAAS

- Enterprise Software application:
Sharing of data between internal and external users e.g. : Salesforce CRM application
- Single user Software application
Runs on single user computer and serves 1 user at a time e.g. : Microsoft office
- Business Utility SaaS - Applications like Salesforce automation are used by businesses and individuals for managing and collecting data, streamlining collaborative processes and providing actionable analysis. Popular use cases are Customer Relationship Management (CRM), Human Resources and Accounting.
- Social Networking SaaS - Applications like Facebook are used by individuals for networking and sharing information, photos, videos, etc.

Consideration for SAAS Application development



Important factors for good design of SAAS model

- Three distinct points that separates a well-design from a poorly designed SAAS application
 - *scalability*
 - *Multi tenant efficient*
 - *configurable*
- Scalability- maximizing concurrency, and efficient use of resources
 - i.e. optimizing locking duration, statelessness, sharing pooled resources such as threads and network connections, caching reference data, and partitioning large databases

Cont..

- *Configurable* - a single application instance on a single server has to accommodate users from several different companies
 - Customizing the application for one customer will change the application for other customers as well.
- Traditionally customizing an application would mean changes in the code.
- Each customer must use metadata to *configure* the way the application appears and behaves for its users.
- Customers configuring applications must be simple and easy without any extra development or operation costs

Cont..

- Multi-tenancy – important architectural shift from designing isolated, single-tenant applications
 - One application instance should accommodate users from multiple other companies at the same time while providing transparency
 - This requires an architecture that maximizes the sharing of resources efficiently across tenants

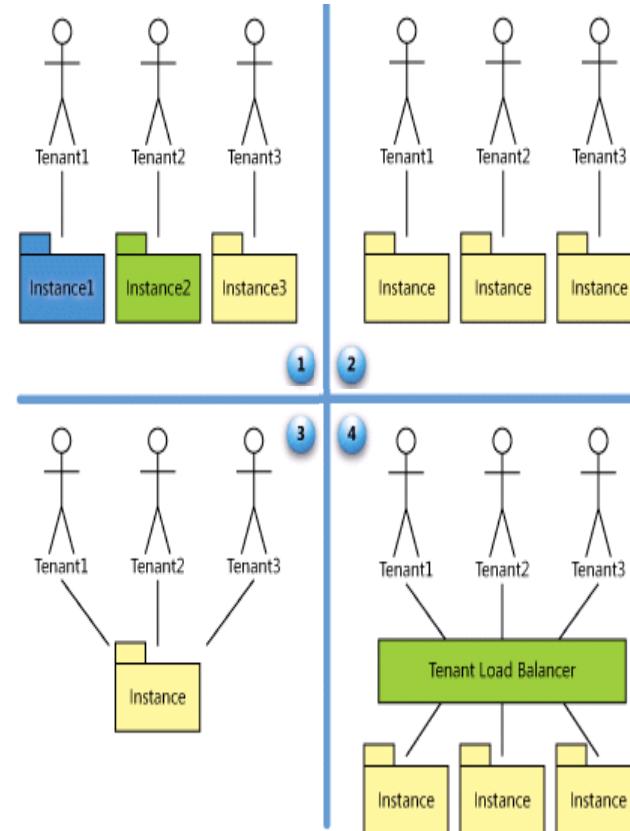
SaaS Maturity Model

Level 1: Ad-Hoc/Custom –
One Instance per customer

Level 2: Configurable
per customer

Level 3: configurable
& Multi-Tenant-
Efficient

Level 4: Scalable,
Configurable & Multi-
Tenant-Efficient



SAAS service providers



Salesforce.com

- Salesforce CRM provides a complete solution for that includes feature-rich solutions for marketing, sales, services, partner management and community management.
- CRM is originally software for managing customer interaction, such as scheduling tasks, emailing, texting, and many more.
- Salesforce grew into a cloud software solution and acquired several other companies for Paas(Platform as a Service) and Saas (Software as a Service).



Salesforce services



Advantages of SaaS

- Easy to use** – Most SaaS applications do not require more than a web browser to run
- Cheap- The pay as you go pricing model of SaaS makes it affordable to small businesses and individuals.
- Scalability**: SaaS application can be easily scaled up or down to meet consumer demand. Consumers do not need to worry about additional computing infrastructure to scale up.
- Applications are less prone to data loss since data is being stored in the cloud.
- Compared to traditional applications, SaaS applications are less clunky. They do not require users to install/uninstall binary code on their machines.
- Due to the delivery nature of SaaS through the internet, SaaS applications are able to run on a wide variety of devices.
- Allows for better collaboration between teams since the data is stored in a central location.
- Velocity of change in SaaS applications is much faster.
- SaaS favors a Agile development life cycle. • Software changes are frequent and on-demand. Most SaaS services are updated about every 2 weeks and users are most time unaware of these changes.

Disadvantages of SaaS

Robustness:

SaaS software may not be as robust (functionality wise) as traditional software applications due to browser limitations. Consider Google Doc & Microsoft Office.

- Privacy

Having all of a user's data sit in the cloud raises security & privacy concerns. SaaS providers are usually the target of hack exploits e.g. Google servers have been the target of exploits purportedly from China in the last several years

Security

Attack detection, malicious code detection

- Reliability

In the rare event of a SaaS provider going down, a wide range of dependent clients could be affected. For example, when Amazon EC2 service went down in April 2011, it took down FourSquare, Reddit, Quora and other well known applications that run on it.

Summary

- SaaS greatly enhances the ability of developers to scale their application on demand and better suite customer needs.
- It encourages Agile practices by enabling providers deliver frequent updates/patches without waiting for major release cycles as in traditional applications.
- SaaS applications however are susceptible to privacy, security and reliability concerns.
- Hybrid environments combining both SaaS and traditional application methodologies may be useful in scenarios of extremely sensitive data or where constant up-time must be maintained.

Homework

- Explain Salesforce in a Layman's term. Why do we need it ?
- Present a brief case study report on services provided by Salesforce.
How is Salesforce different than traditional CRM solutions.

References

- <https://jsarraf.com/ppt/cloud-saas.pptx>



Software Engineering (CSPE41)

Summer 2023

Lecture 8:

Requirements Engineering

Slides based on various web sources including Pressman's text

Instructor: C. Oswald

Topics covered

- ⑩ Functional and non-functional requirements
- ⑩ The software requirements document
- ⑩ Requirements specification
- ⑩ Requirements engineering processes
- ⑩ Requirements elicitation and analysis
- ⑩ Requirements validation
- ⑩ Requirements management

Requirements Engineering

- ⑩ The process of establishing the services that the customer requires from a system and the constraints under which it operates and is developed.

- ⑩ The requirements themselves are the **descriptions** of the system services and **constraints** that are generated during the requirements engineering process.

What is a Requirement?

- ⑩ It may range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification.
- ⑩ This is inevitable as requirements may serve a dual function
 - ⑩ May be the basis for a bid for a contract - therefore must be open to interpretation;
 - ⑩ May be the basis for the contract itself - therefore must be defined in detail;
 - ⑩ Both these statements may be called requirements.

Requirements Abstraction (Davis)

“If a company wishes to let a contract for a large software development project, it must **define its needs** in a sufficiently abstract way that a solution is not pre-defined.

The requirements must be clearly written so that several contractors can bid for the contract, offering, perhaps, different ways of meeting the client organization’s needs.

Once a contract has been awarded, the contractor must write a system definition for the client in more detail so that the client understands and can validate what the software will do.

Both of these documents may be called the requirements document for the system.”

Types of Requirements

⑩ User requirements

- ⑩ Statements in natural language plus diagrams of the services the system provides and its operational constraints.
- ⑩ Written for customers.

⑩ System requirements

- ⑩ A structured document setting out detailed descriptions of the system's functions, services and operational constraints.
- ⑩ Defines what should be implemented so may be part of a contract between client and contractor.
- ⑩ Whom do you think these are written for?
- ⑩ 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, negotiate different approaches, and specify a preliminary set of solution requirements

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**)
 - Exciting requirements: These requirements are for features that go beyond the customer's expectations and prove to be very satisfying when present (e.g. **For a new mobile phone**)

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
- Any prototypes developed to better define requirements

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.

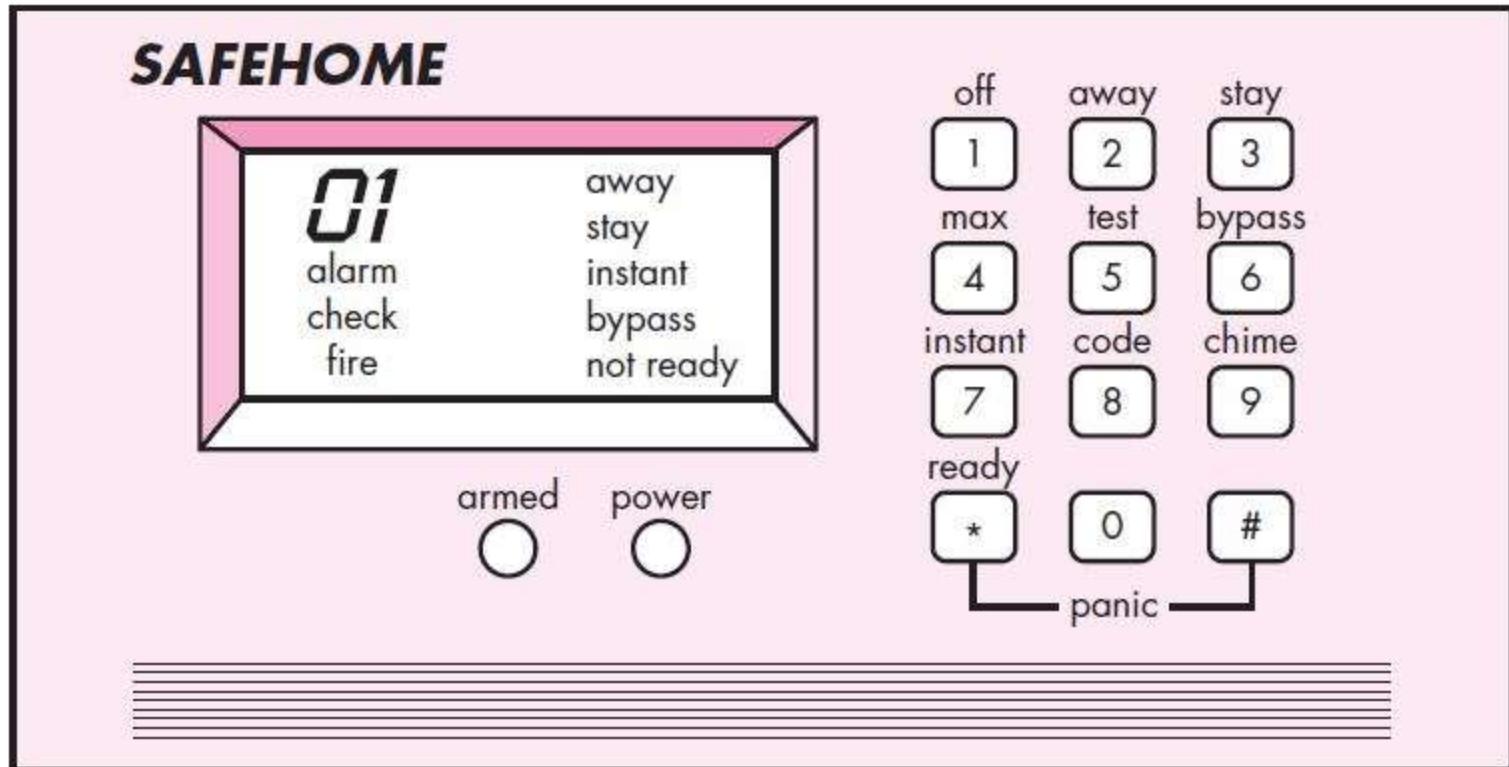
SafeHome actors

- **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 red alarm light to indicate that *SafeHome* has been armed

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?
3. How much time does the homeowner have to enter the password from the time the first key is pressed?

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:

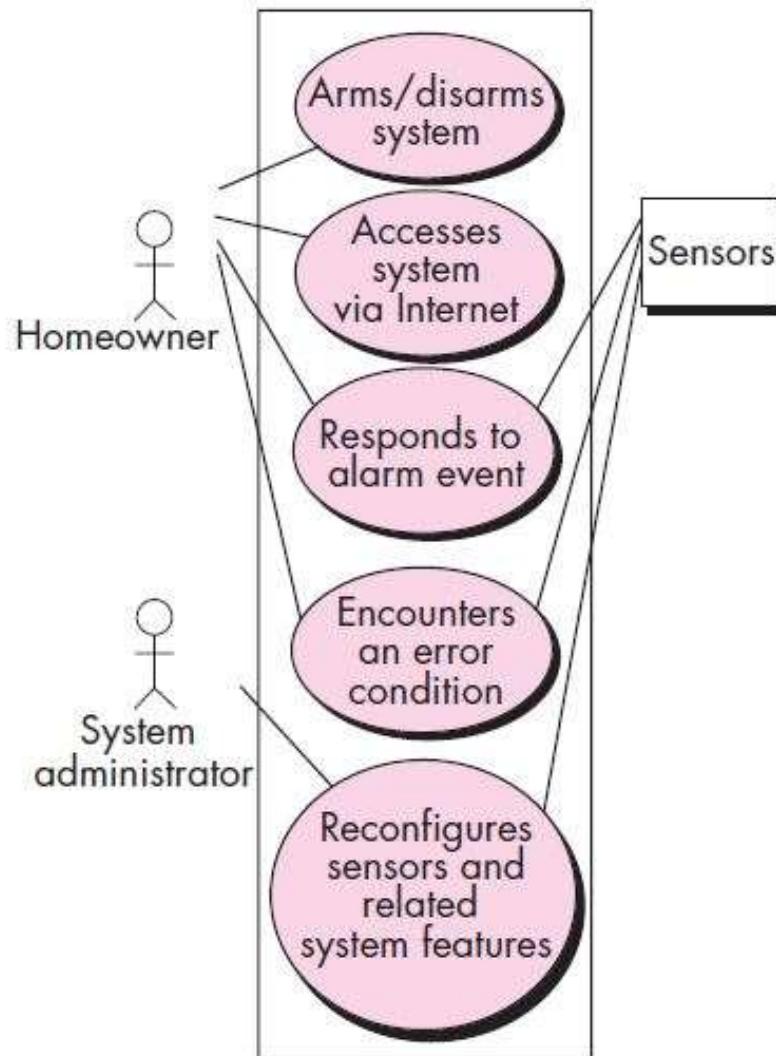
<https://www.ibm.com/support/pages/ibm-rational-rose-enterprise-7004-ifix001>

Others:

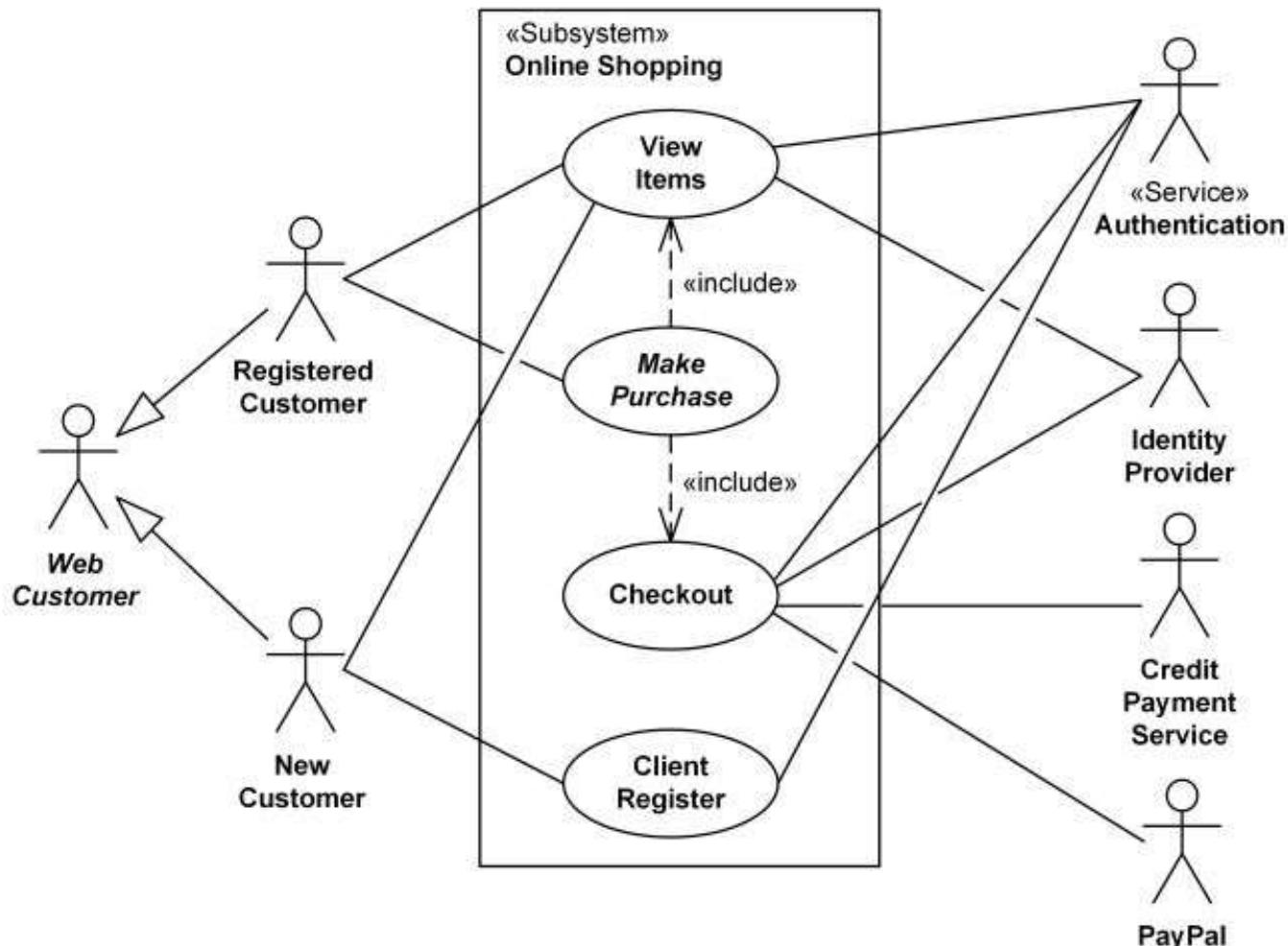
Lucidchart, Microsoft Visio Pro



UML use case diagram for *SafeHome* home security function

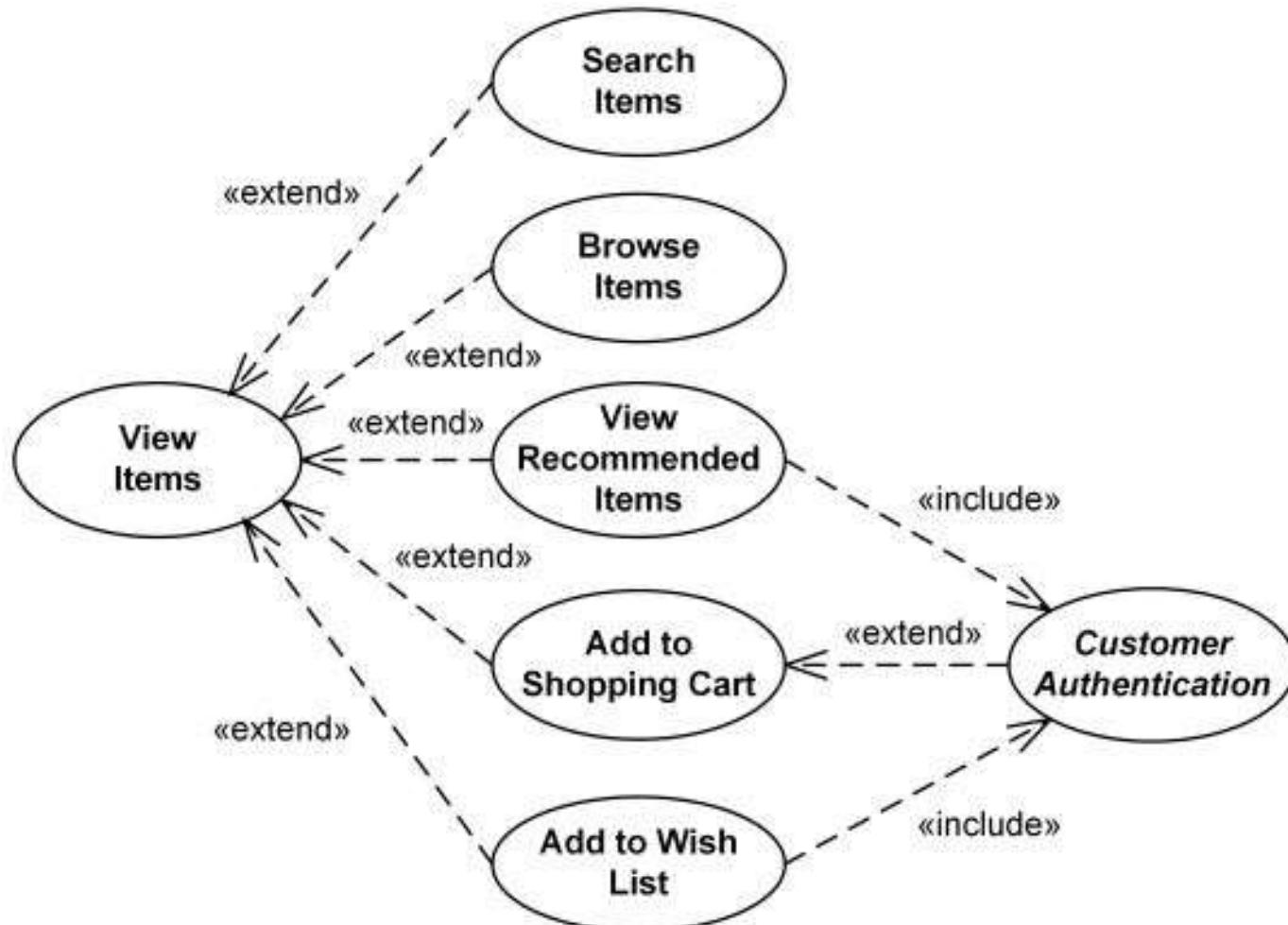


Usecase diagram for Online Shopping Tool

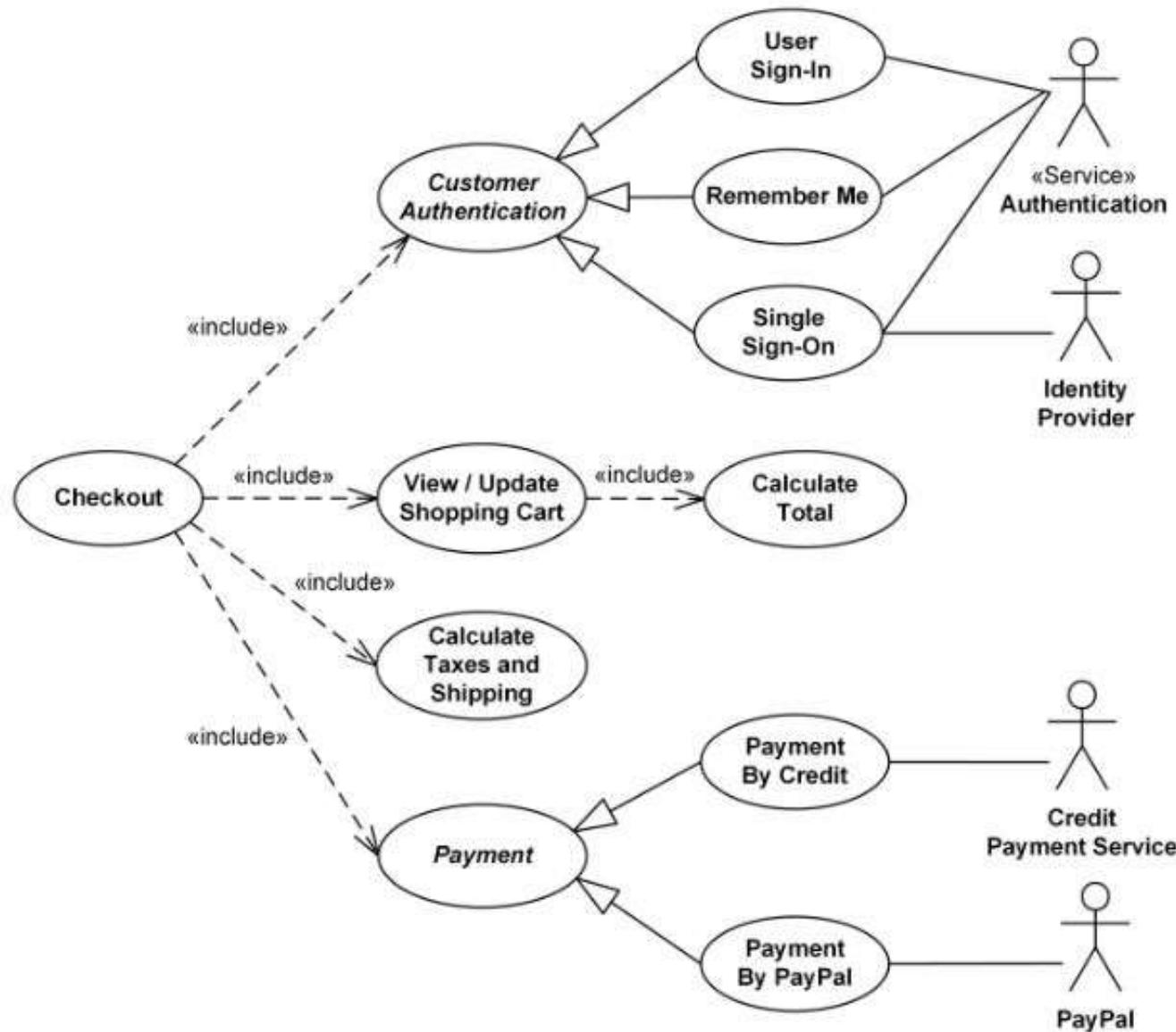


<https://www.uml-diagrams.org/examples/online-shopping-use-case-diagram-example.html>

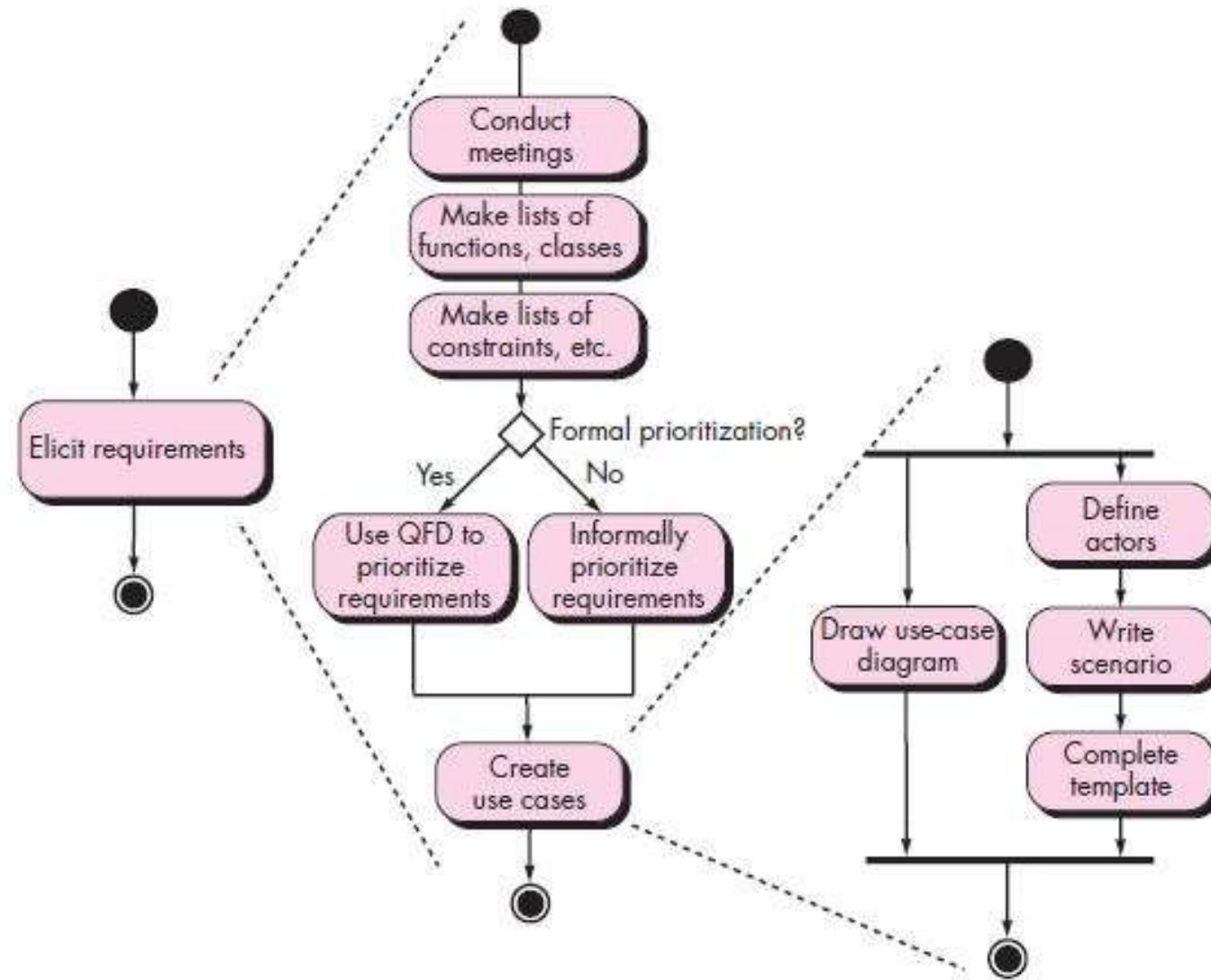
Online shopping use case diagram example - view items use case.



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



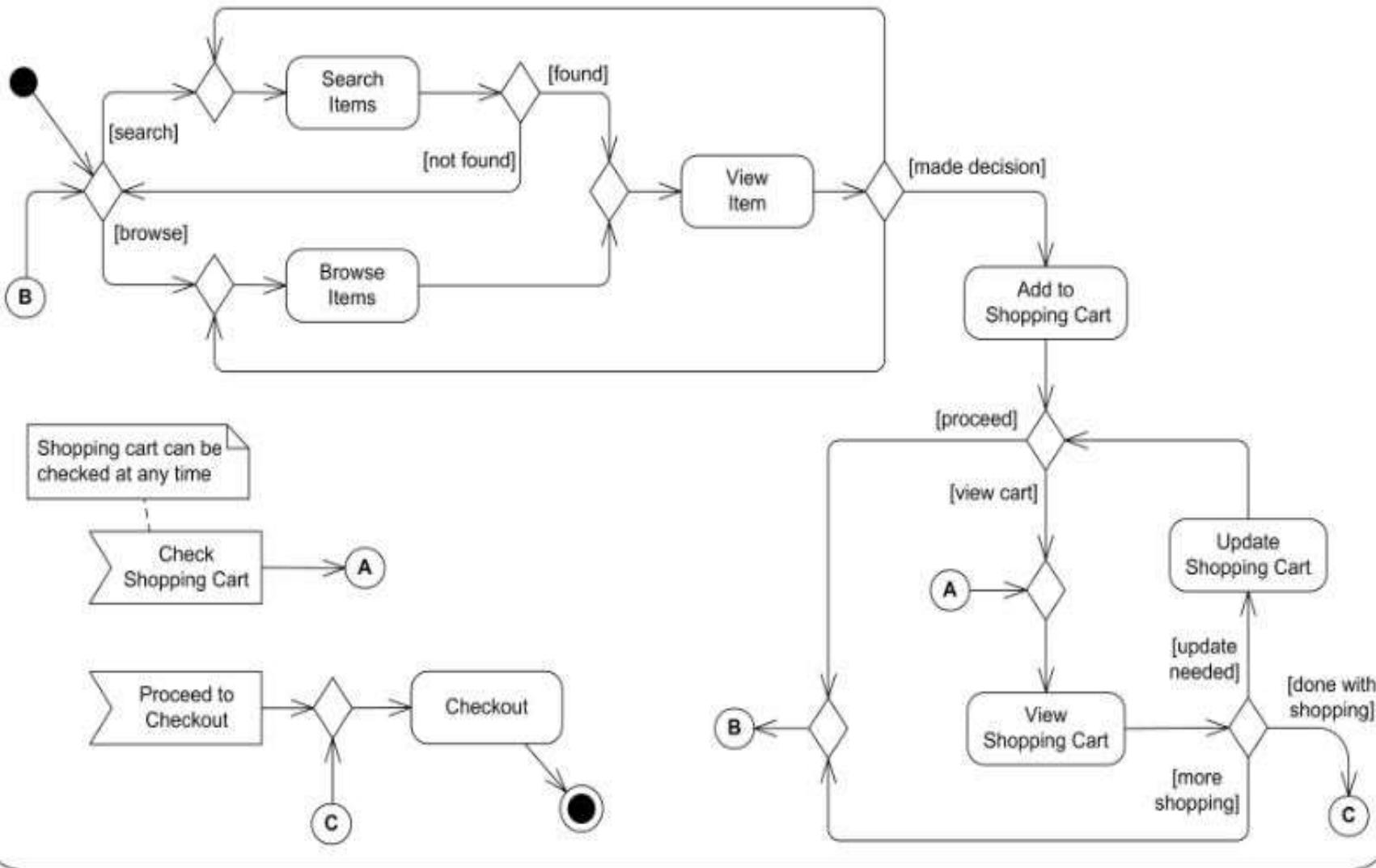
UML activity diagrams for eliciting requirements



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 changes?

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 comes into a system, what functions are applied to that data to do transformations, and what resulting output data are produced

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

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

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

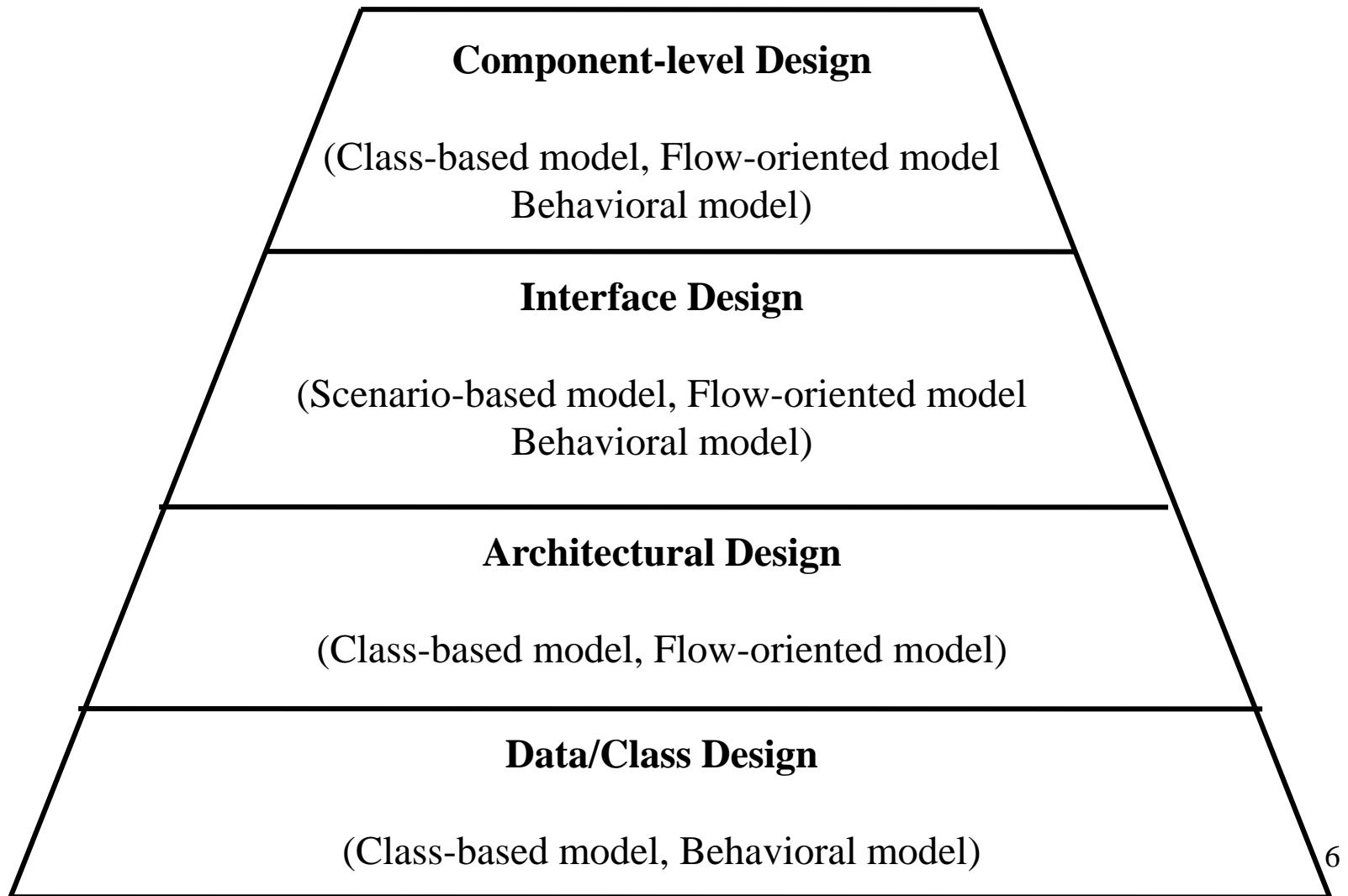
The Process of Design

- 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*

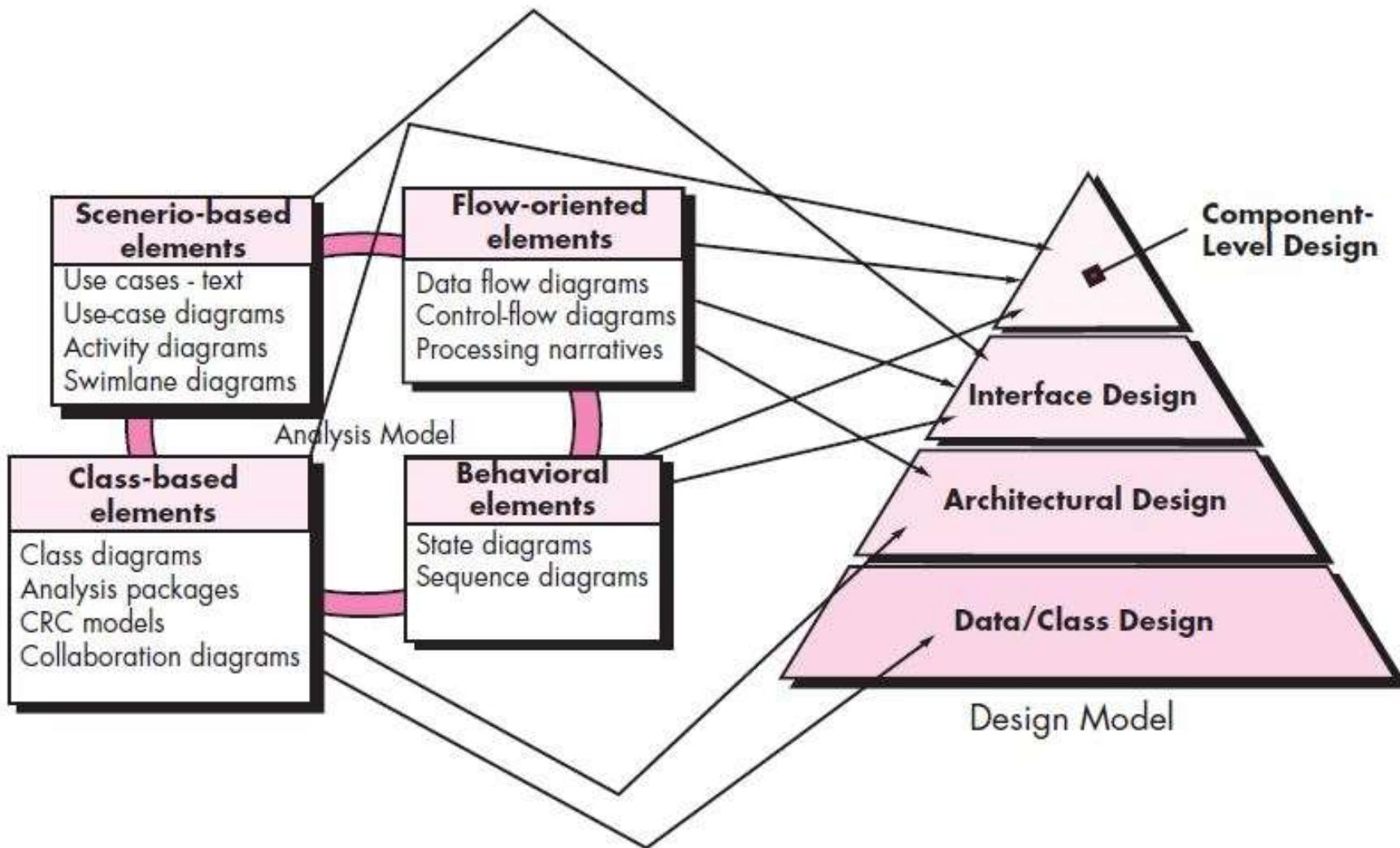
Why design?

- 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

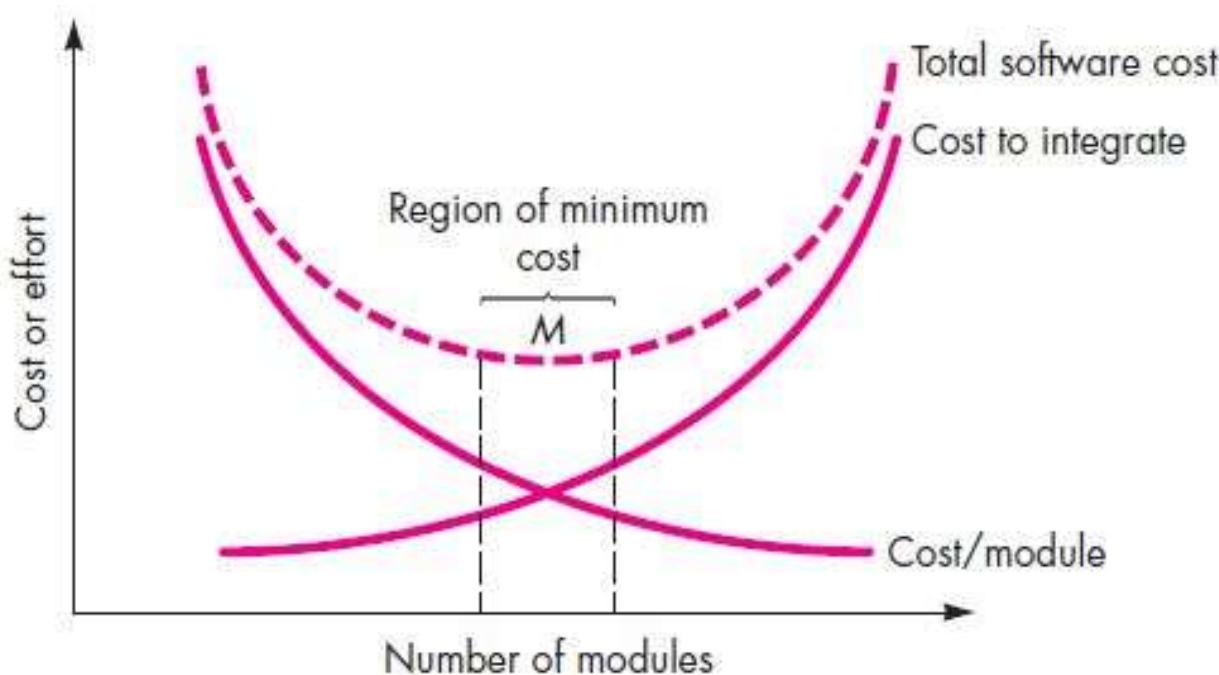
- suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently

concern is a feature or behavior that is specified as part of the requirements model for the software.

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

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 the business solution

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.

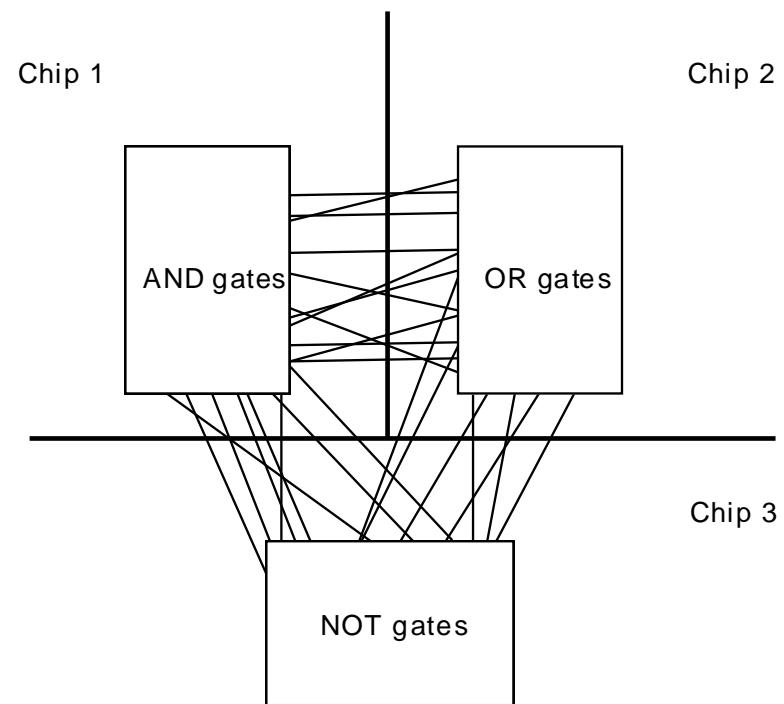
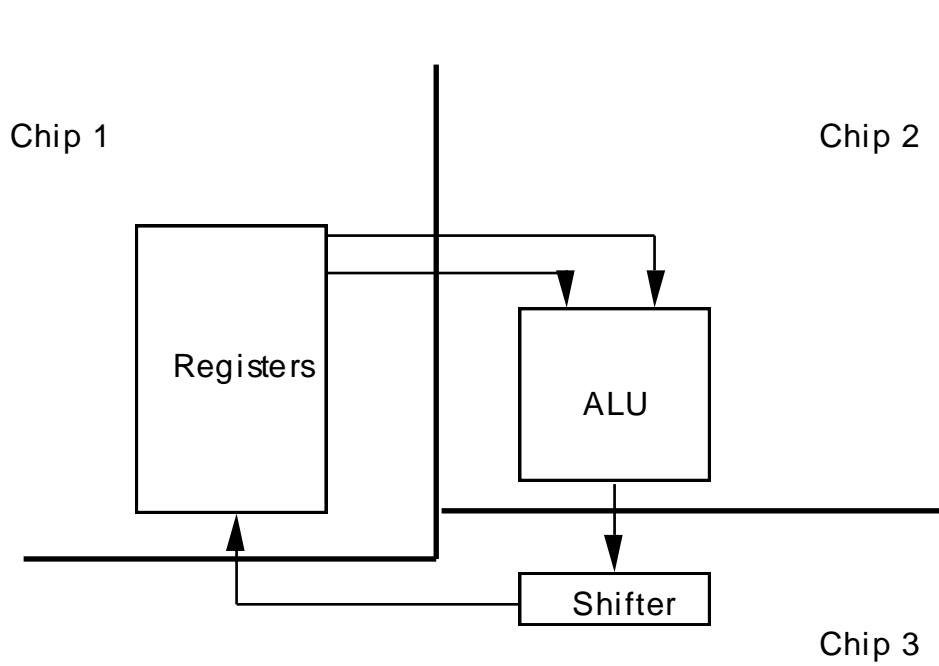
- Requirement *A* - enable a registered user to access video from cameras placed throughout a space
- Requirement *B* - *registered user must be validated prior to using the website*
- *A** is a design representation for requirement *A* and *B** is a design representation for requirement *B*.
- Therefore, *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 unrelated functions) are to be avoided.

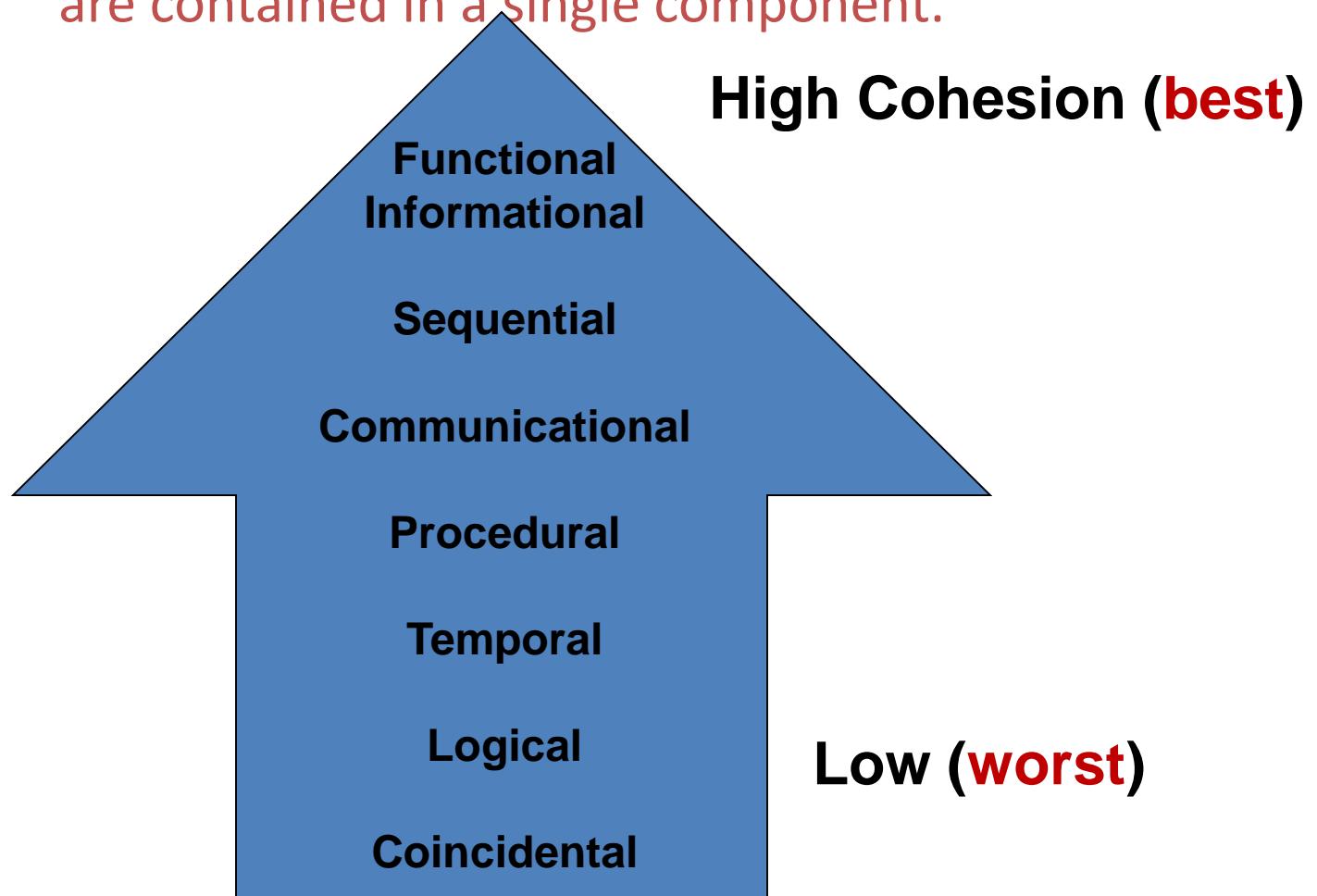
Good vs. Bad Modules

- 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.



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-class components. The tape sub-class reads from tape. The disk sub-class reads from disk. The network sub-class reads from the network.

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 () .
```

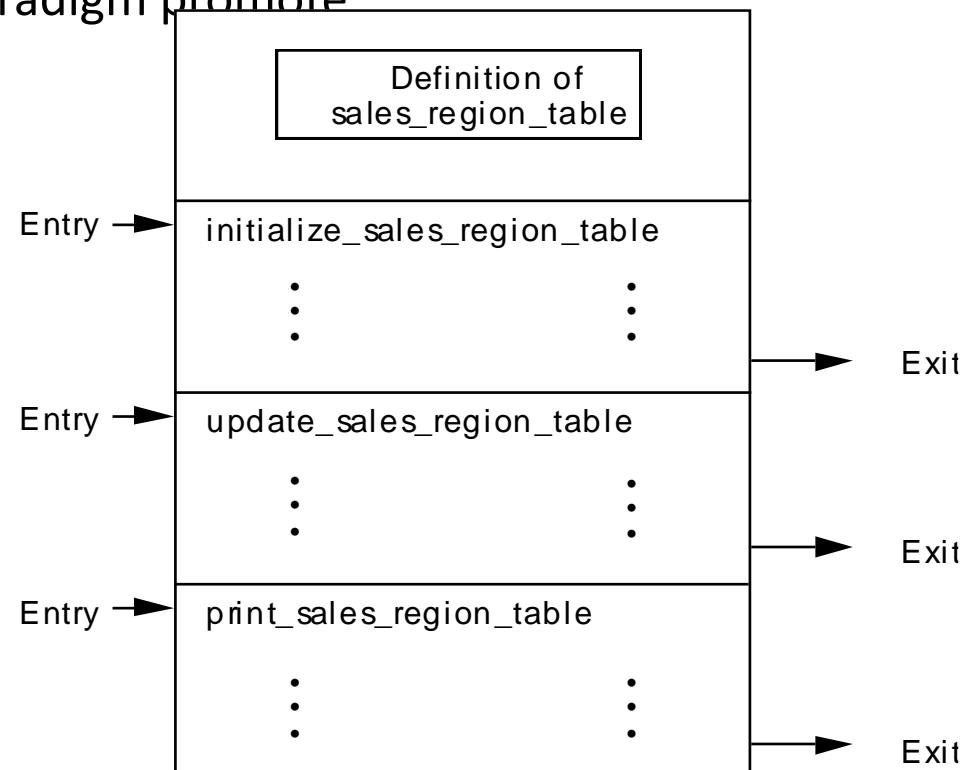
- Example 2
 - calculate new coordinates and send them to window**
- Why is this bad?
 - still leads to less reusability -> break it up

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

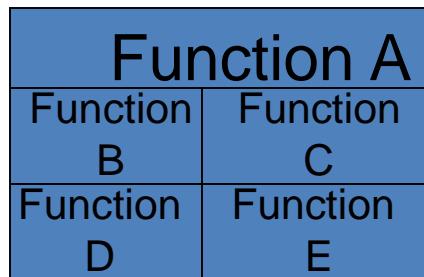


7. Functional Cohesion

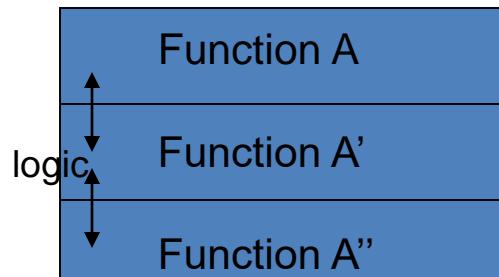
- Def.
 - module performs exactly one action
- Examples
 - get temperature of day
 - compute orbital of electron
 - calculate sales commission
- Why is this good?
 - more reusable
 - corrective maintenance easier
 - fault isolation
 - reduced regression faults
 - easier to extend product

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

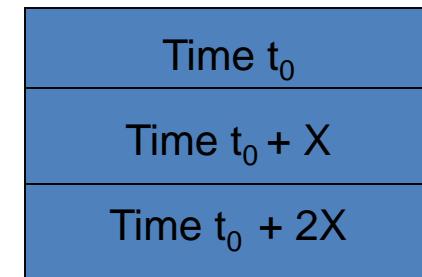
Examples of Cohesion-1



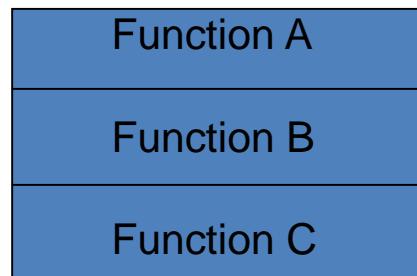
Coincidental
Parts unrelated



Logical
Similar functions

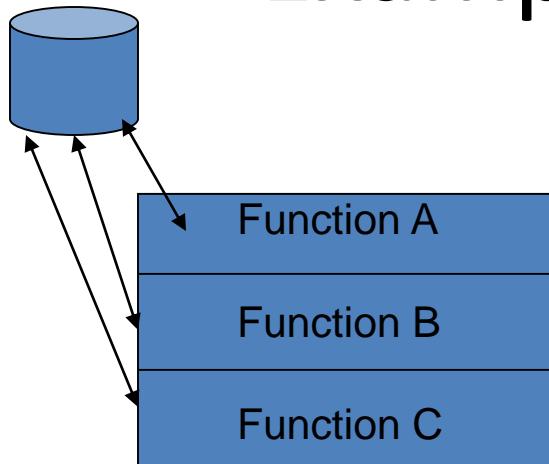


Temporal
Related by time

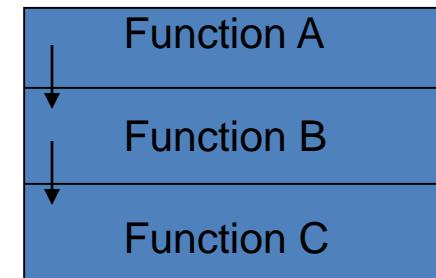


Procedural
Related by order of functions

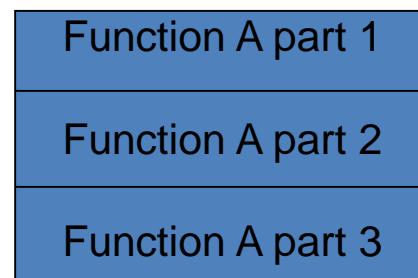
Examples of Cohesion-2



Communicational
Access same data



Sequential
Output of one is input to another

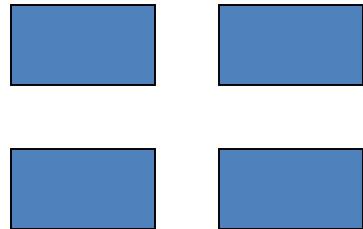


Functional
Sequential with complete, related functions

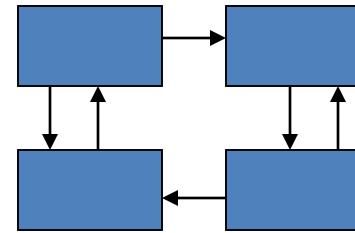
Study

- 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.

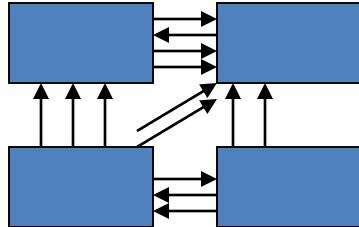
Coupling: Degree of dependence among components



No dependencies



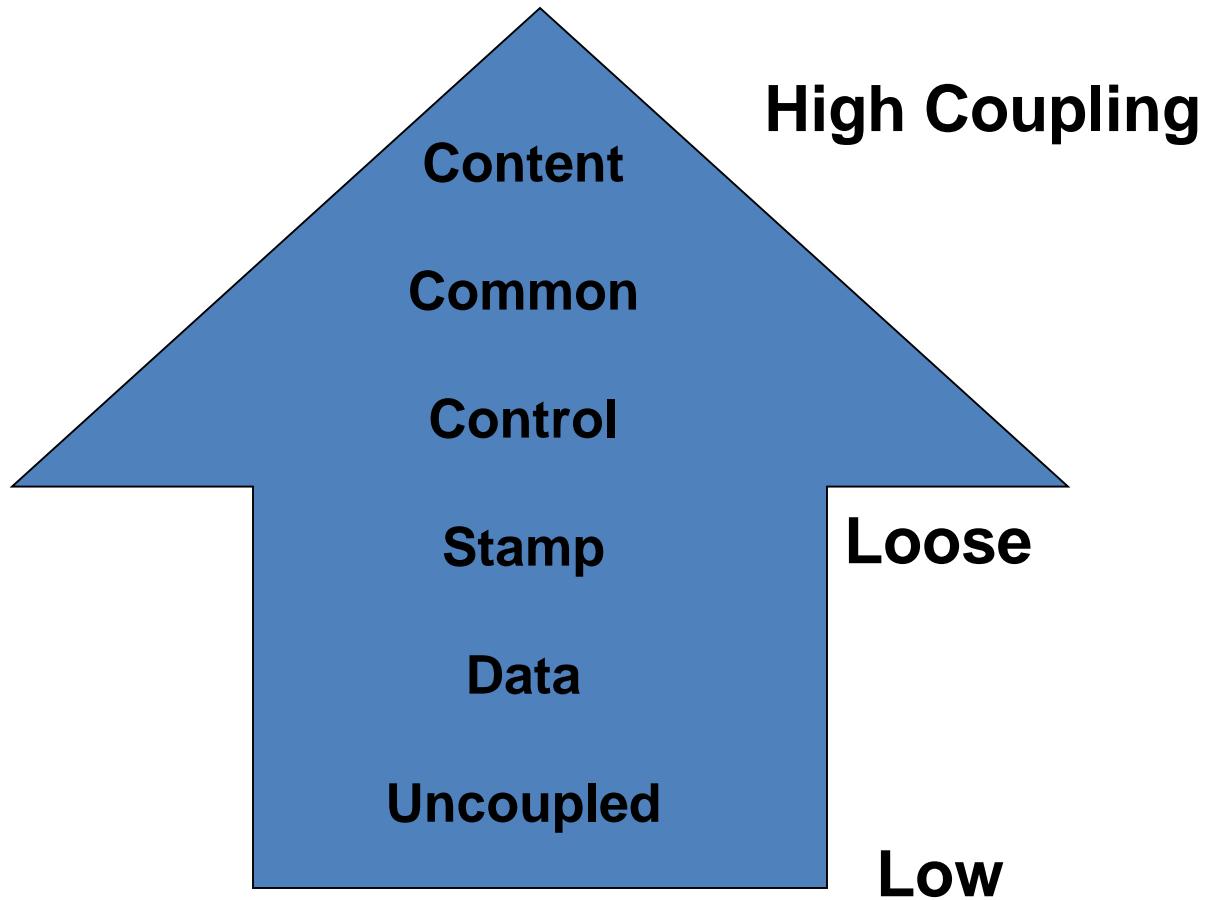
Loosely coupled-some dependencies



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.

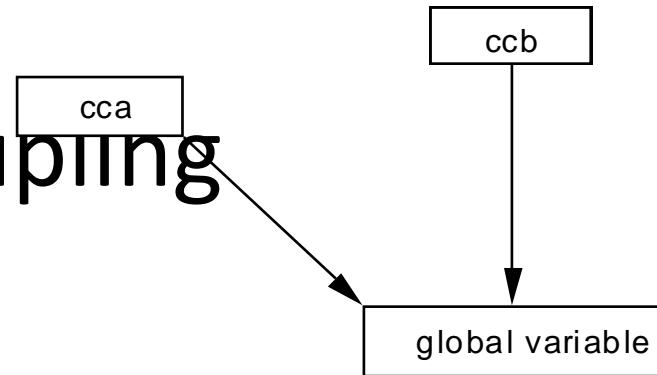
When customer not found, component adds customer by directly modifying the contents of the data structure containing customer data.

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.

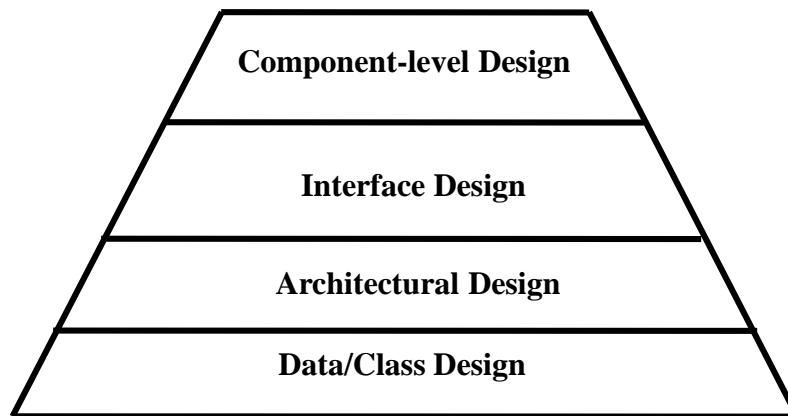
5. Data Coupling

- 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

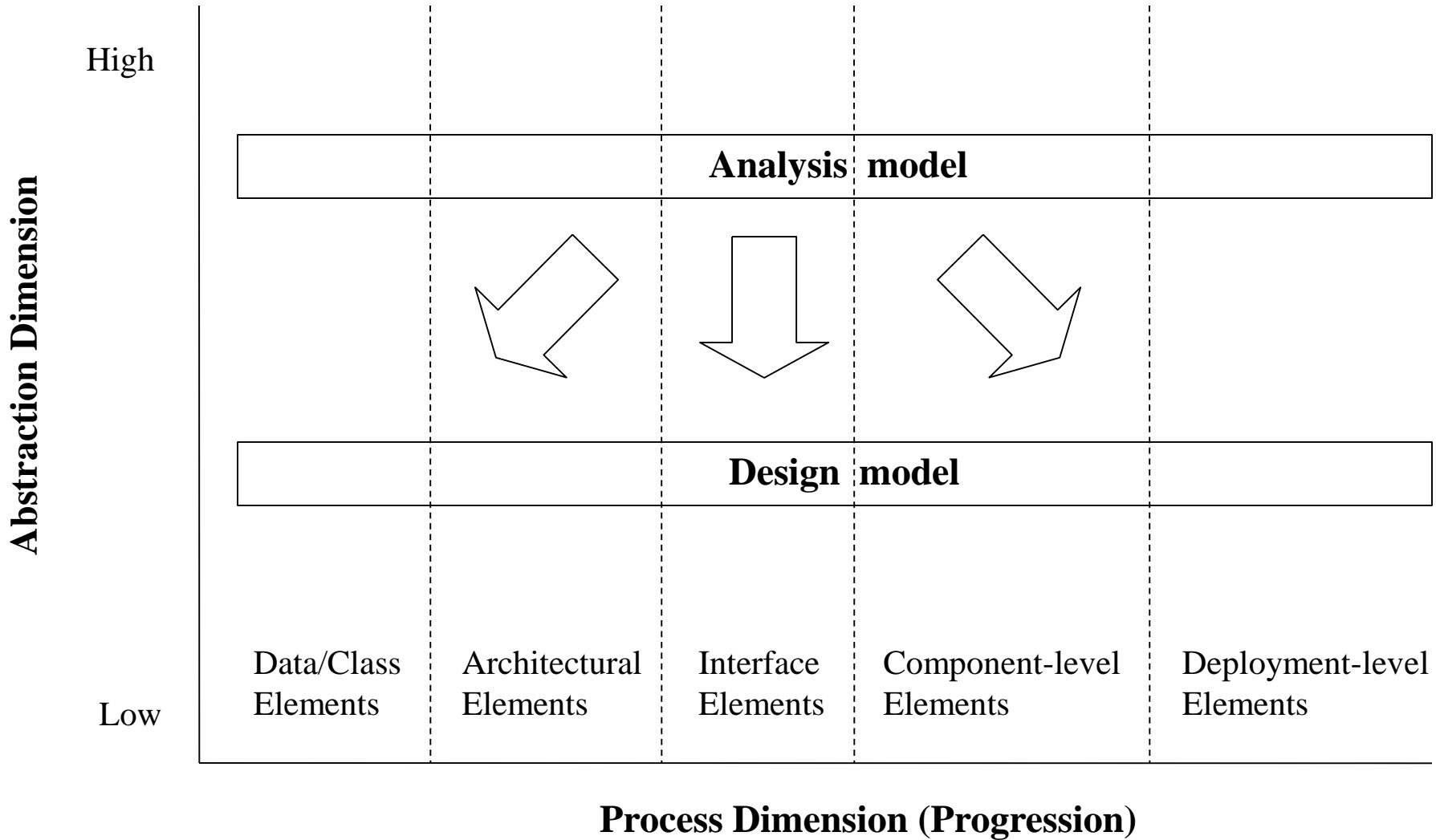
6. No Coupling

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

The Design Model



Dimensions of the Design Model

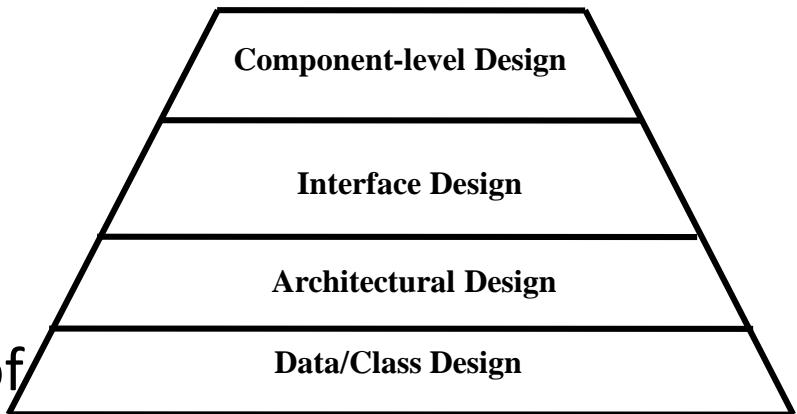


Introduction

- The design model can be viewed in two different dimensions
 - (Horizontally) The process dimension indicates the evolution of the parts of the design model as each design task is executed
 - (Vertically) The abstraction dimension represents the level of detail as each element of the analysis model is transformed into the design model and then iteratively refined
- Elements of the design model use many of the same UML diagrams used in the analysis model
 - The diagrams are refined and elaborated as part of the design
 - More implementation-specific detail is provided
 - Emphasis is placed on
 - Architectural structure and style
 - Interfaces between components and the outside world
 - Components that reside within the architecture

Introduction (continued)

- Design model elements are not always developed in a sequential fashion
 - Preliminary architectural design sets the stage
 - It is followed by interface design and component-level design, which often occur in parallel
- The design model has the following layered elements
 - Data/class design
 - Architectural design
 - Interface design
 - Component-level design
- A fifth element that follows all of the others is deployment-level design



Design Elements

- Data/class design
 - Creates a model of data and objects that is represented at a high level of abstraction
- Architectural design
 - Depicts the overall layout of the software
- Interface design
 - Tells how information flows into and out of the system and how it is communicated among the components defined as part of the architecture
 - Includes the user interface, external interfaces, and internal interfaces
- Component-level design elements
 - Describes the internal detail of each software component by way of data structure definitions, algorithms, and interface specifications
- Deployment-level design elements
 - Indicates how software functionality and subsystems will be allocated within the physical computing environment that will support the software

Types of Design Classes

- **User interface classes** – define all abstractions necessary for human-computer interaction (usually via metaphors of real-world objects)
- **Business domain classes** – refined from analysis classes; identify attributes and services (methods) that are required to implement some element of the business domain
- **Process classes** – implement business abstractions required to fully manage the business domain classes
- **Persistent classes** – represent data stores (e.g., a database) that will persist beyond the execution of the software
- **System classes** – implement software management and control functions that enable the system to operate and communicate within its computing environment and the outside world

Characteristics of a Well-Formed Design Class

- Complete and sufficient
 - Contains the complete encapsulation of all attributes and methods that exist for the class
 - Contains only those methods that are sufficient to achieve the intent of the class
- Primitiveness
 - Each method of a class focuses on accomplishing one service for the class
- High cohesion
 - The class has a small, focused set of responsibilities and single-mindedly applies attributes and methods to implement those responsibilities
- Low coupling
 - Collaboration of the class with other classes is kept to an acceptable minimum
 - Each class should have limited knowledge of other classes in other subsystems

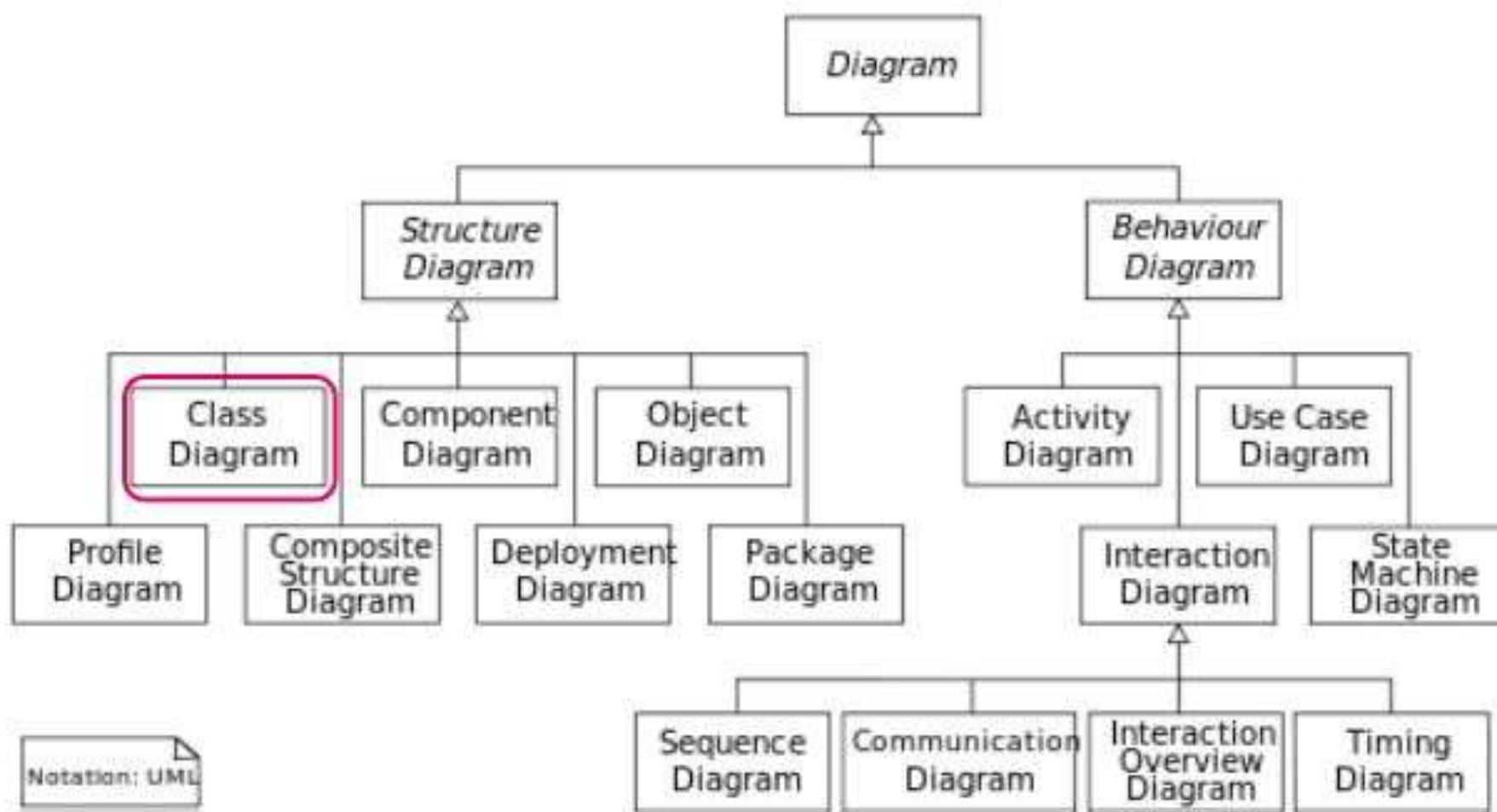
Pattern-based Software Design

- Mature engineering disciplines make use of thousands of design patterns for such things as buildings, highways, electrical circuits, factories, weapon systems, vehicles, and computers
- Design patterns also serve a purpose in software engineering
- Architectural patterns
 - Define the overall structure of software
 - Indicate the relationships among subsystems and software components
 - Define the rules for specifying relationships among software elements
- Design patterns
 - Address a specific element of the design such as an aggregation of components or solve some design problem, relationships among components, or the mechanisms for effecting inter-component communication
 - Consist of creational, structural, and behavioral patterns
- Coding patterns
 - Describe language-specific patterns that implement an algorithmic or data structure element of a component, a specific interface protocol, or a mechanism for communication among components

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



Notation: UML

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

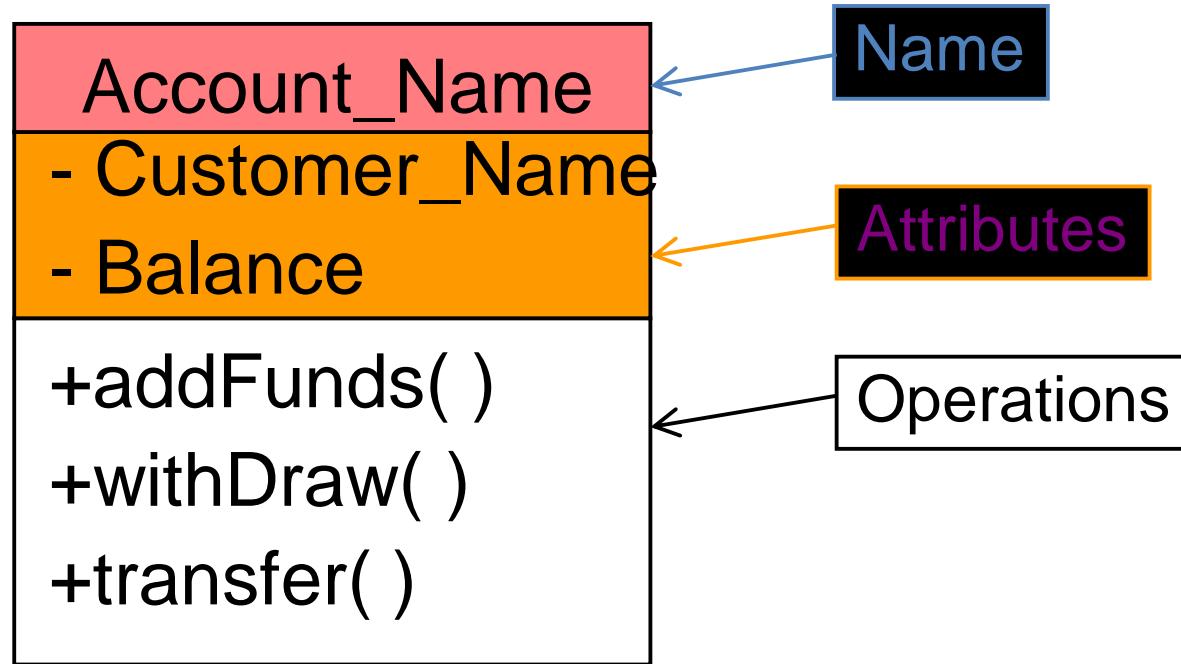
- Each class is represented by a rectangle subdivided into three compartments
 - Name
 - Attributes
 - Operations
- Modifiers are used to indicate visibility of attributes and operations.
 - ‘+’ is used to denote *Public* visibility (everyone)
 - ‘#’ is used to denote *Protected* visibility (friends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

Access Specifiers

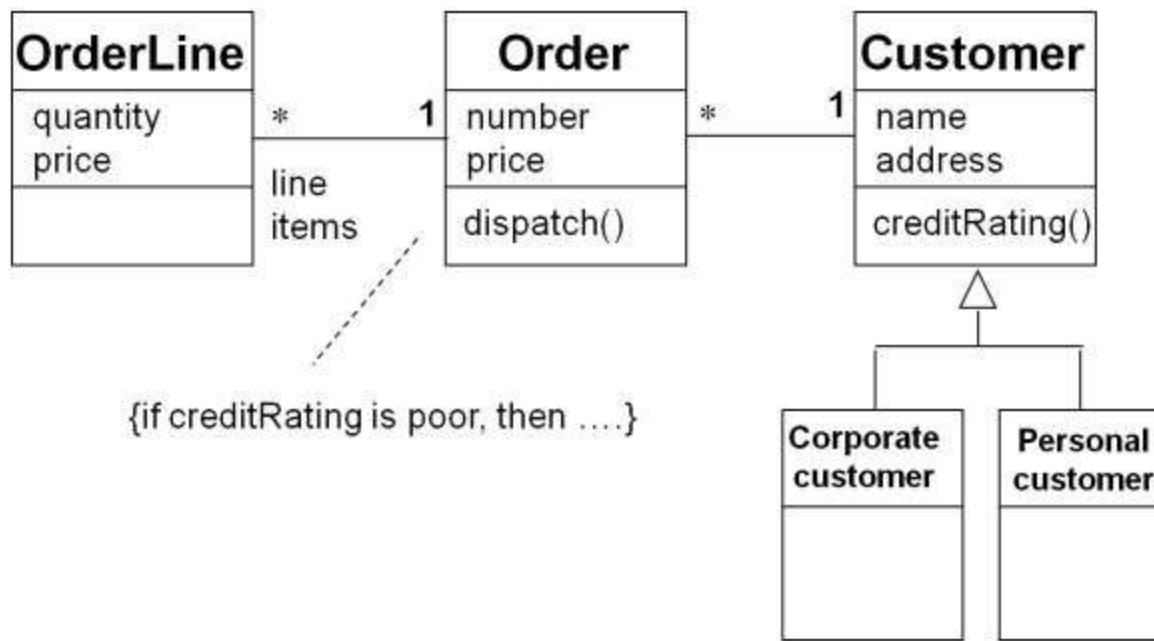
Access for each of these visibility types is shown below for members of different classes

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

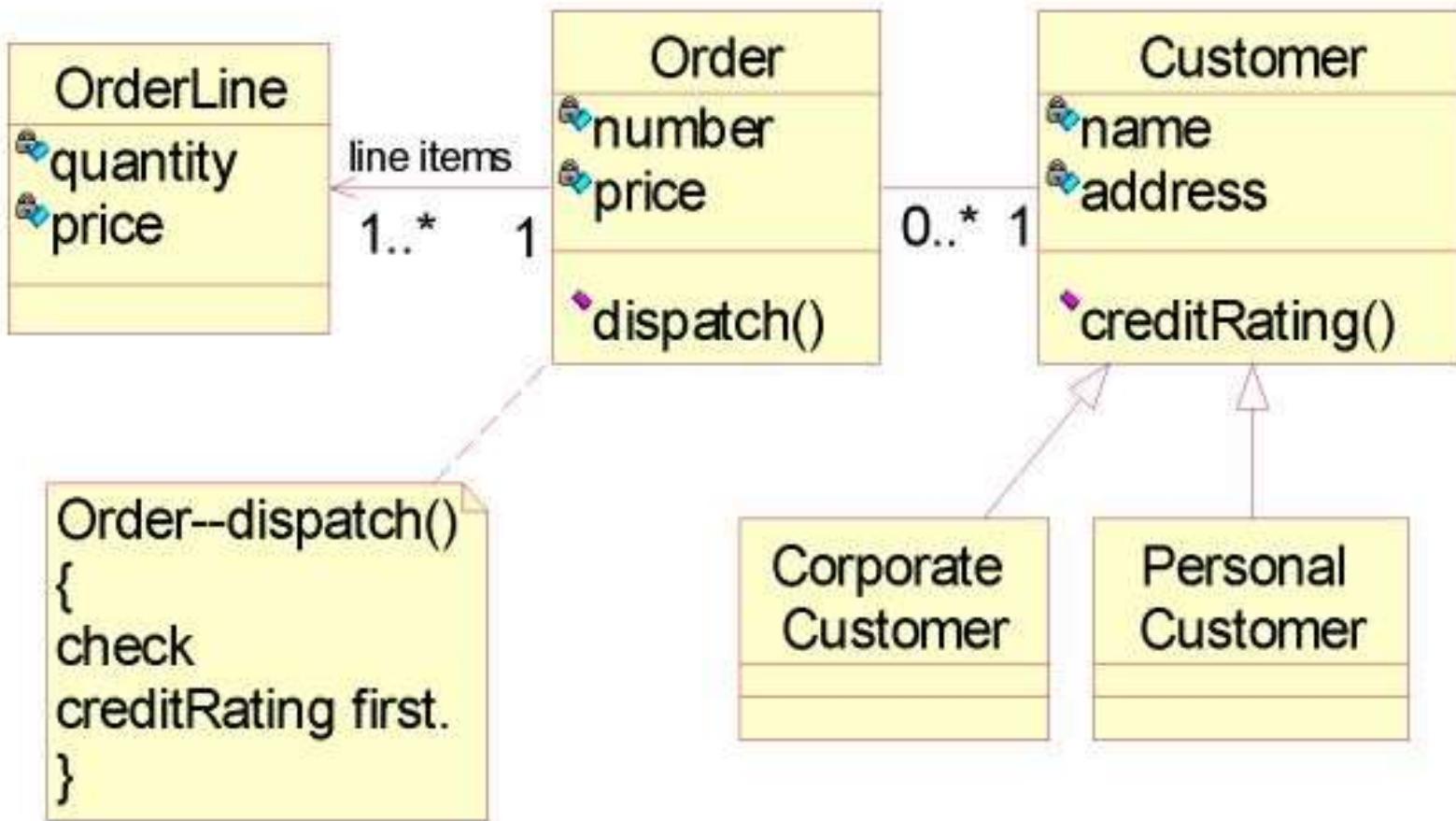
Class diagram



Class Diagram: Example



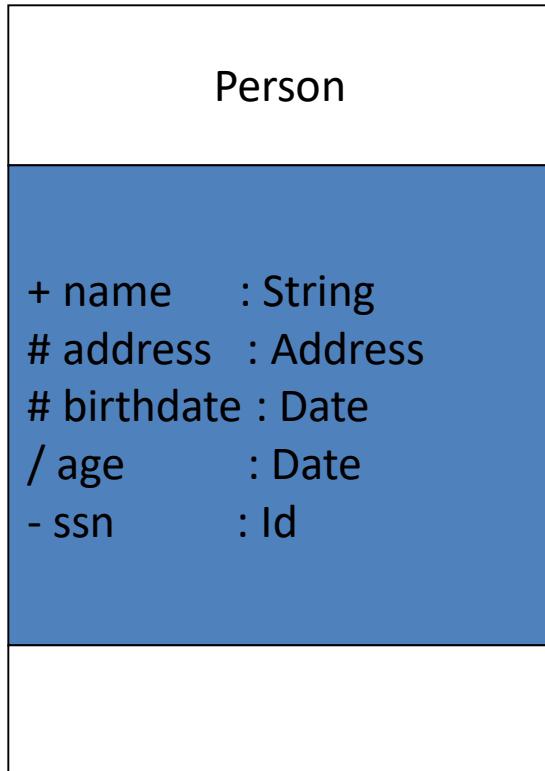
Rational Rose - Example of a class diagram



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)

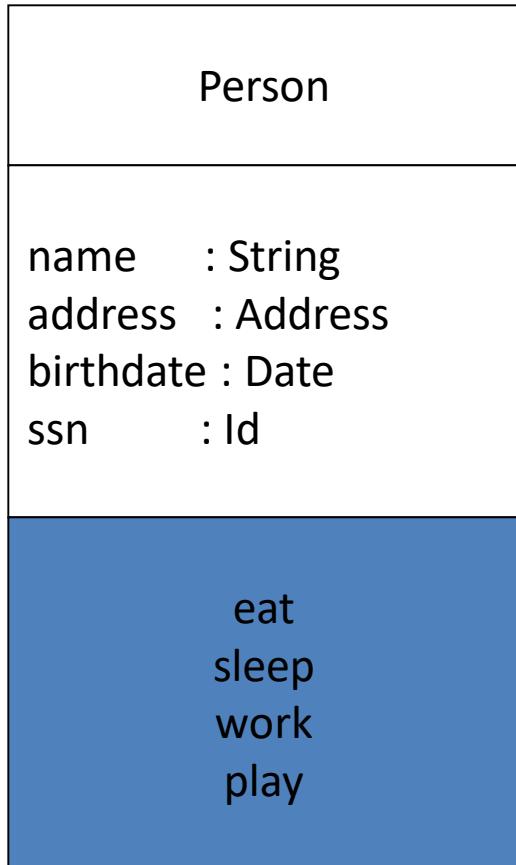
Class Attributes (Cont'd)



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

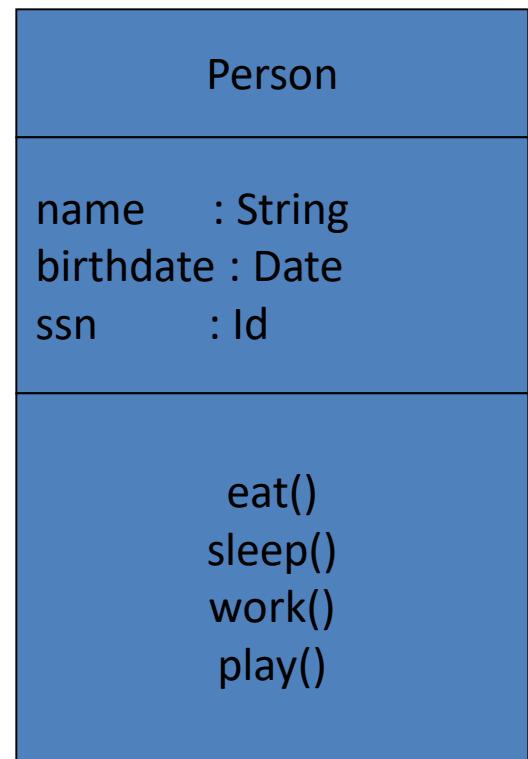
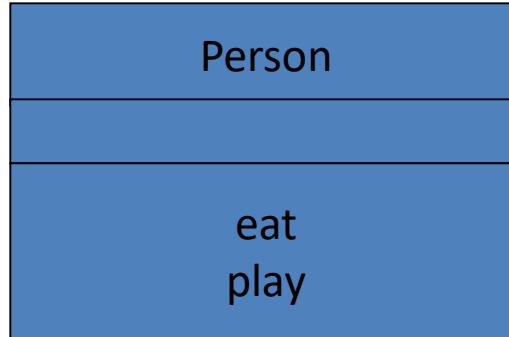
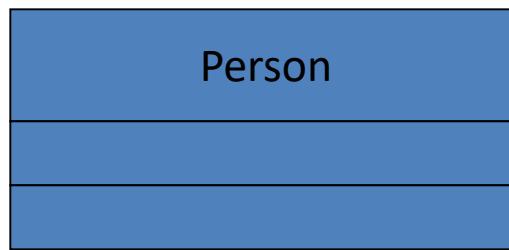
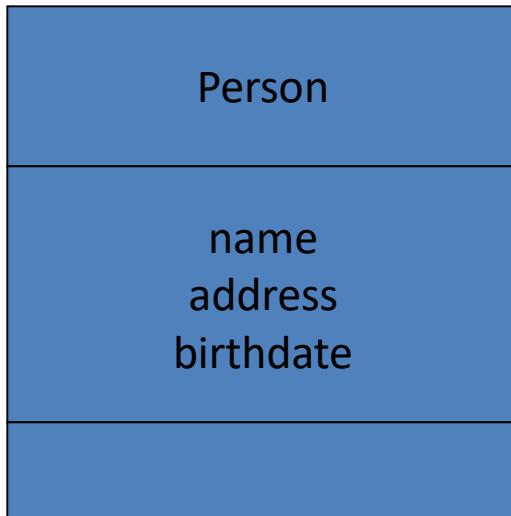
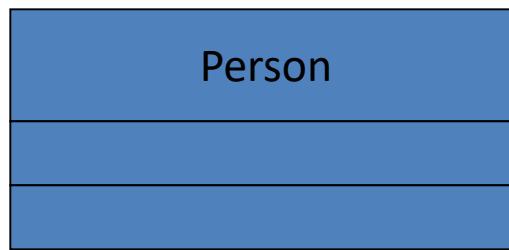
Exercise – Class Identification

- Identify meaningful classes and attributes in the:

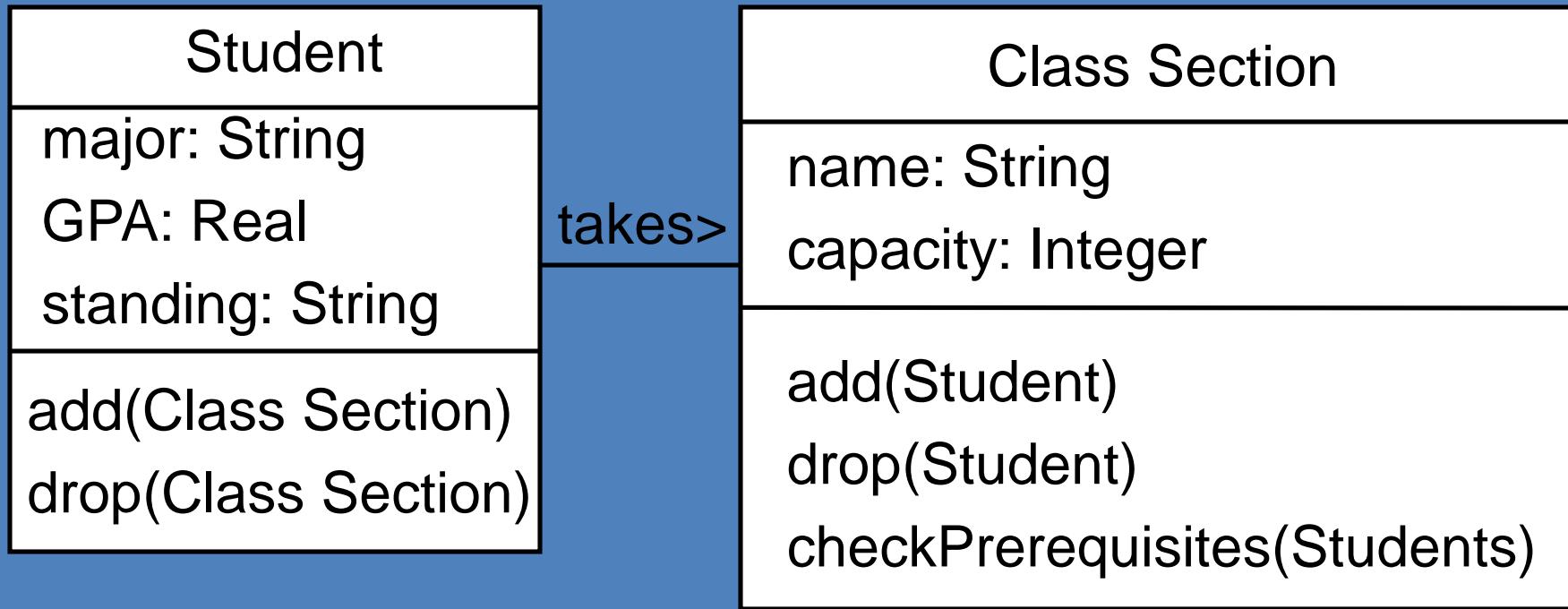
SafeHome System

Depicting Classes

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



Operations



Course

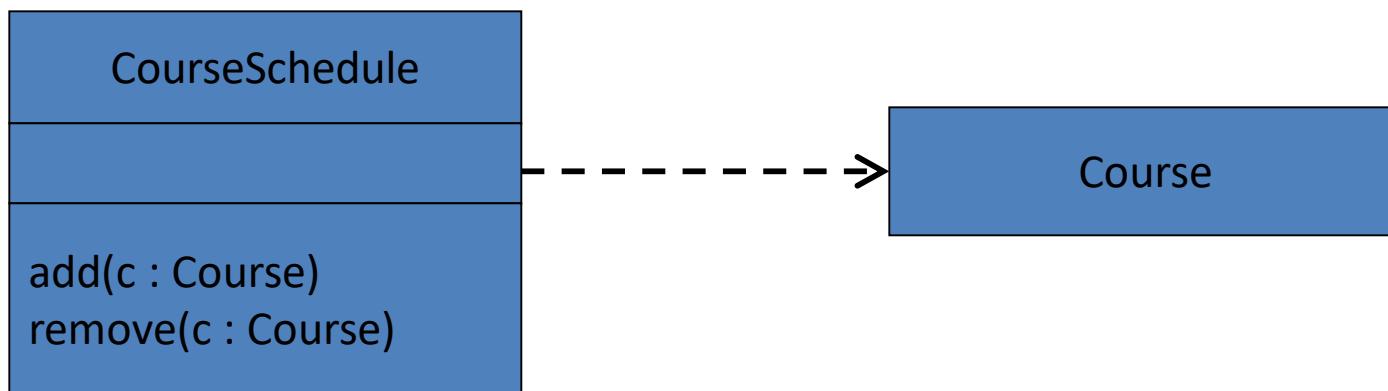
Prerequisite

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

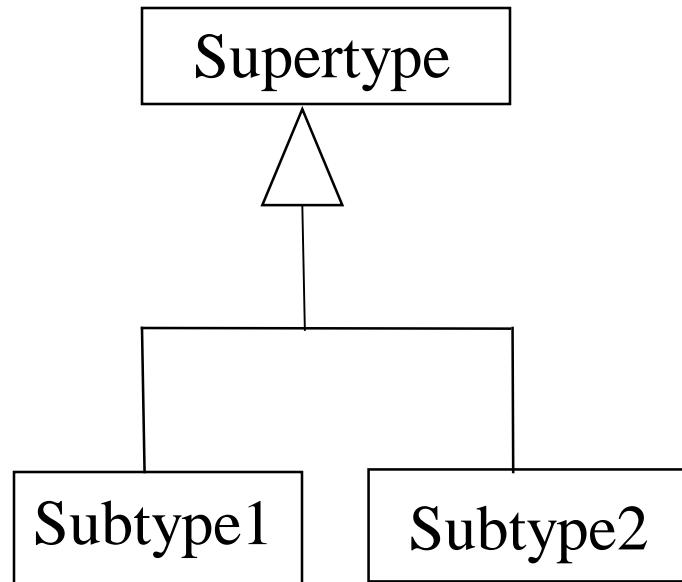
A *dependency* indicates a semantic relationship between two or more elements. The dependency from *CourseSchedule* to *Course* exists because *Course* is used in both the **add** and **remove** operations of *CourseSchedule*.



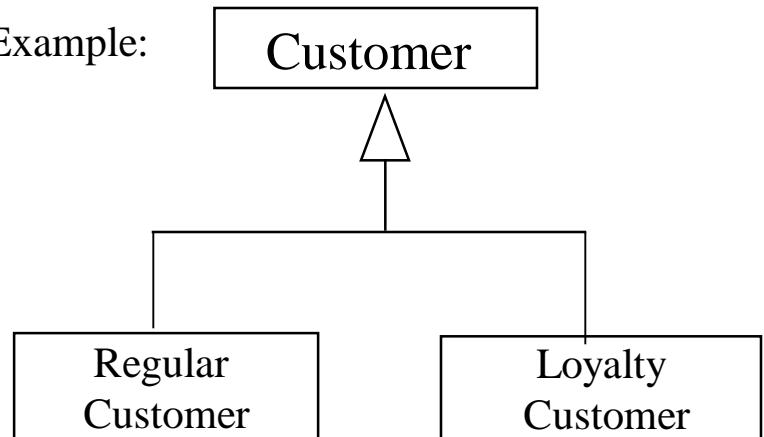
Dependencies

- 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



Example:

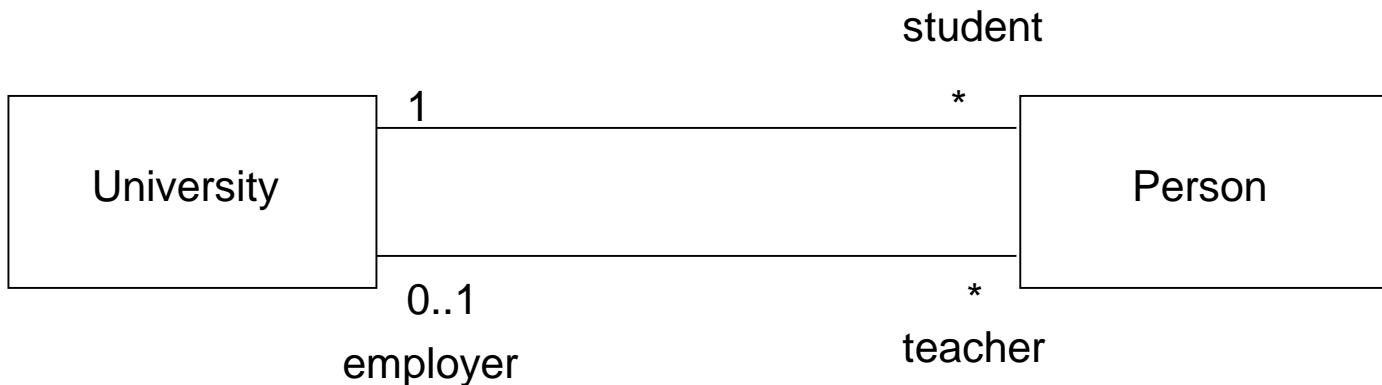


- Inheritance is a required feature of object orientation
- Generalization expresses a parent/child relationship among related classes.
- Used for abstracting details in several layers

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

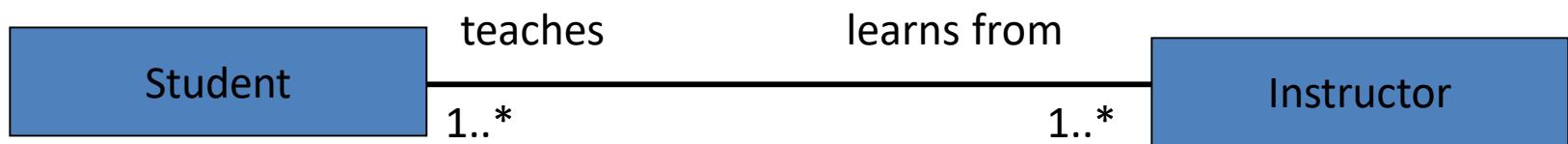
Symbol	Meaning
1	One and only one
0..1	Zero or one
M..N	From M to N (natural language)
*	From zero to any positive integer
0..*	From zero to any positive integer
1..*	From one to any positive integer

Role

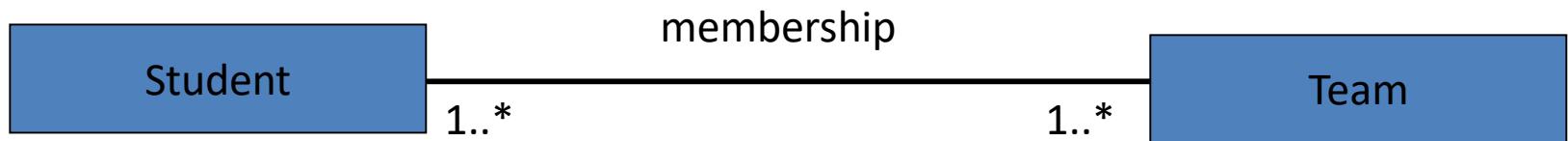
"A given university groups many people; some act as students, others as teachers. A given student belongs to a single university; a given teacher may or may not be working for the university at a particular time."

Association Relationships (Cont'd)

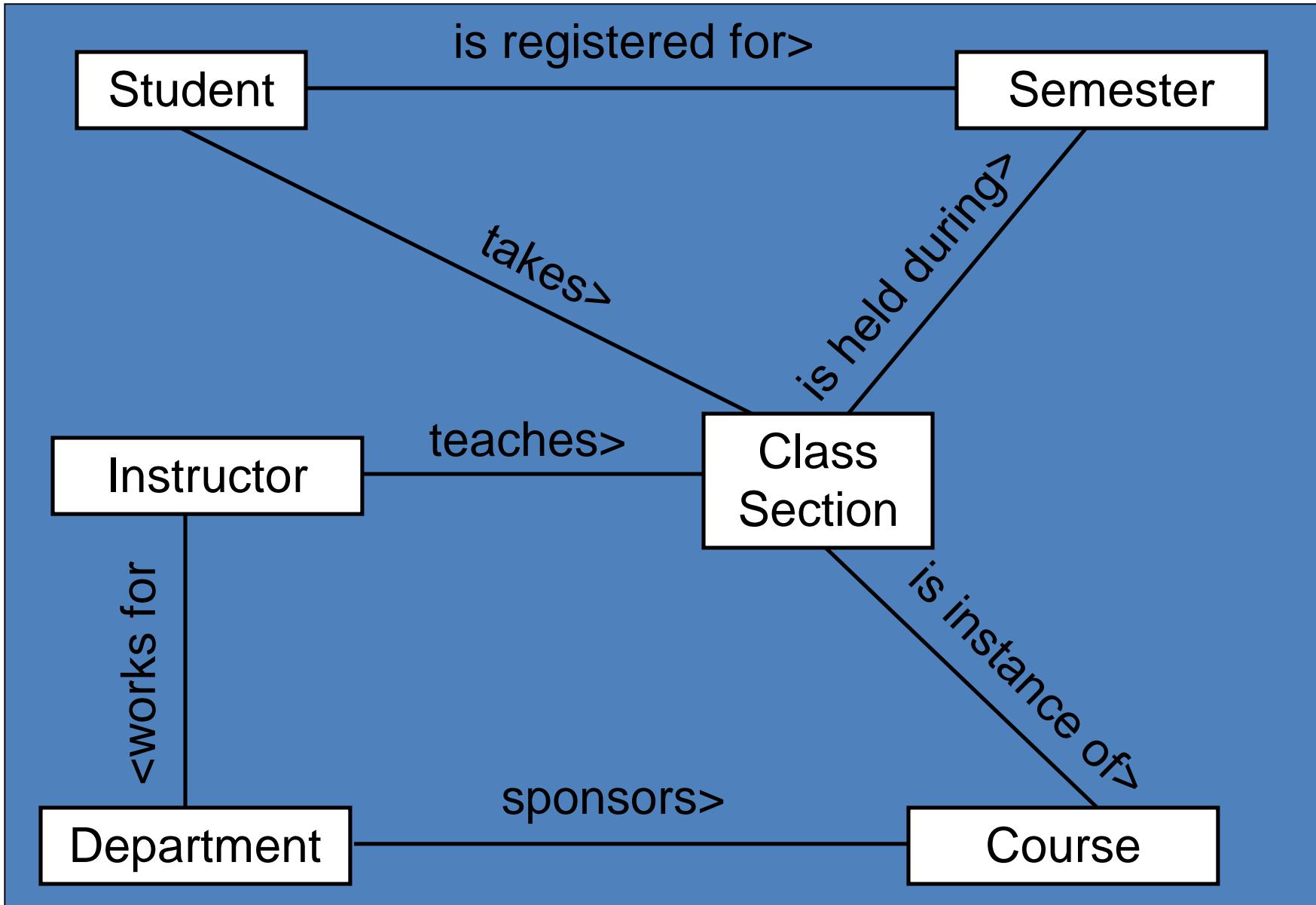
We can also indicate the behavior of an object in an association (*i.e.*, the *role* of an object) using *rolenames*.



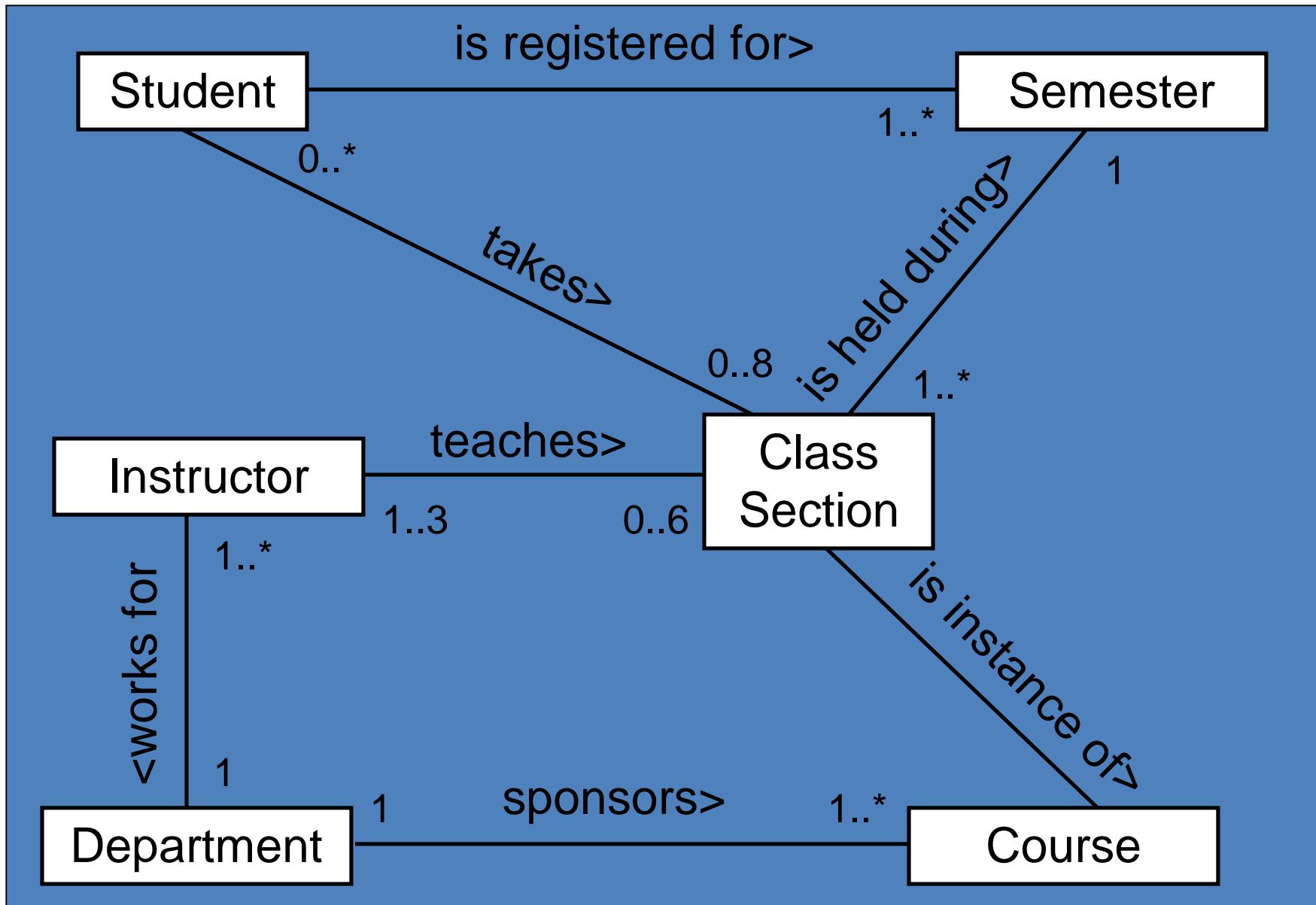
We can also name the association



Associations



Multiplicity Constraints



Questions

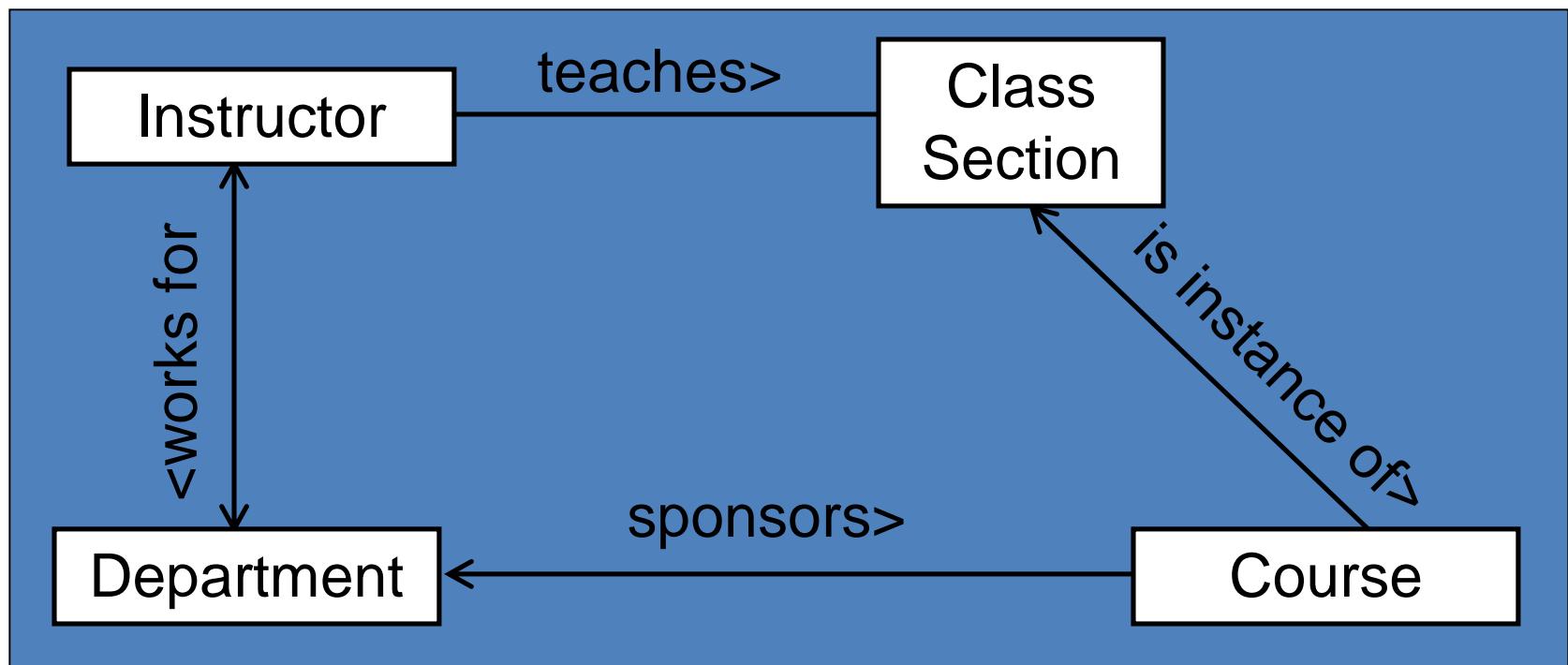
- From the previous diagram
 - How many classes can a student take?
 - Do you have to be registered in any classes to be a student?
 - Do I need to teach this class to be an Instructor?
 - Do I need to teach ANY classes?

Association Names

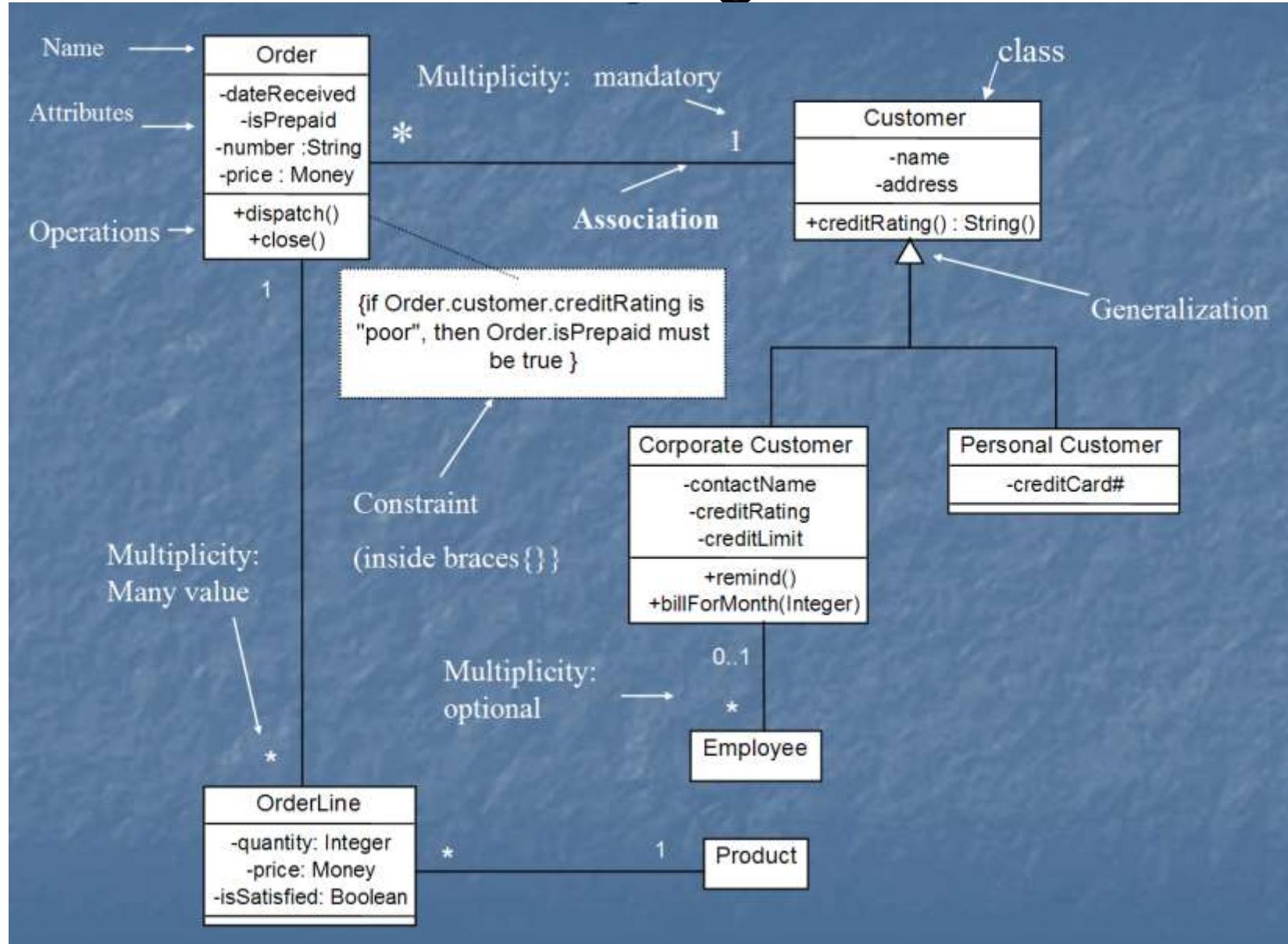
- Associations may be named
 - The names should communicate the meaning of the links
 - The names are typically verb phrases
 - The name should include an arrow indicating the direction in which the name should be read

Navigation

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



Class diagram



[from *UML Distilled Third Edition*]

Association: Model to Implementation

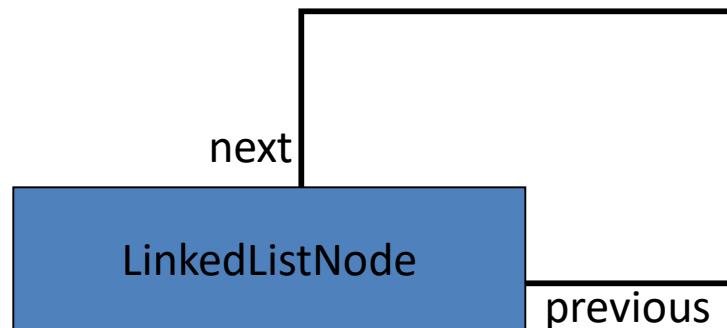


```
Class Student {  
    Course enrolls[4];  
}
```

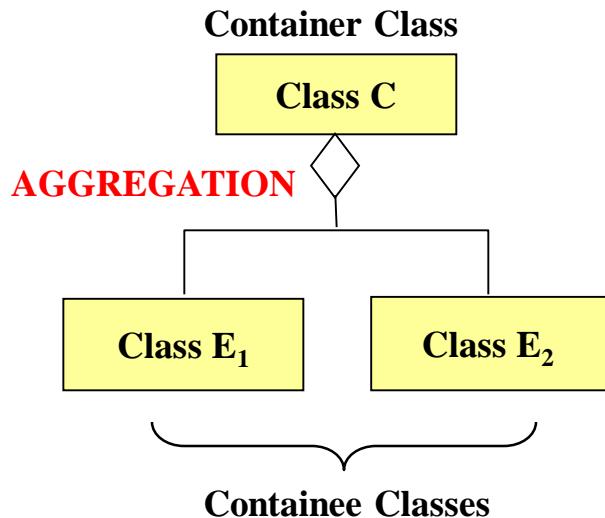
```
Class Course {  
    Student have[];  
}
```

Association Relationships (Cont'd)

A class can have a *self association*.



OO Relationships: Aggregation



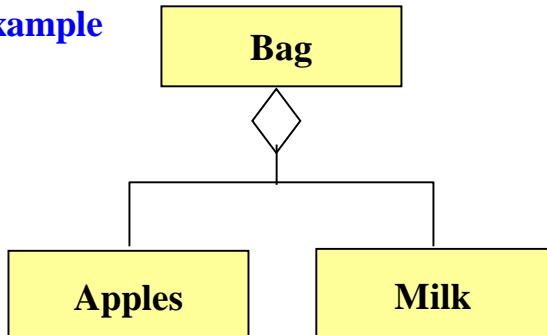
Aggregation:

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.

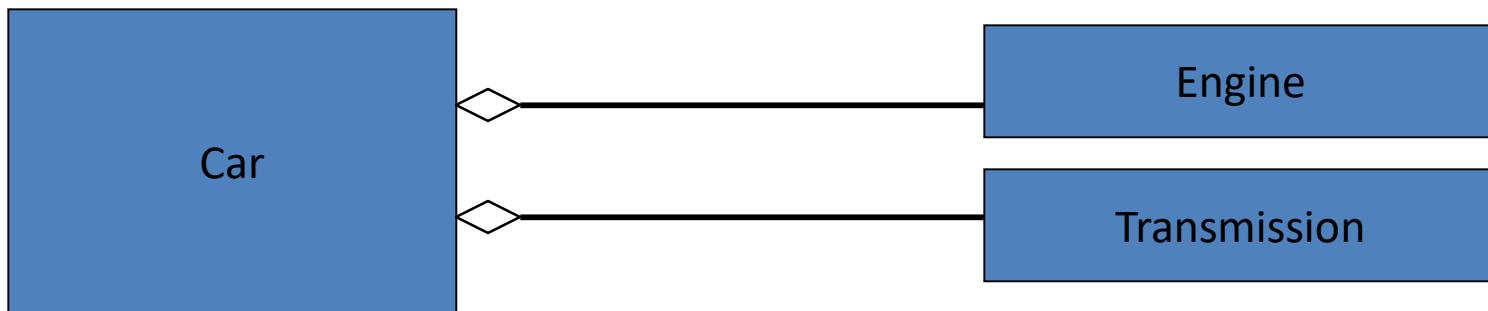
Example



Aggregation

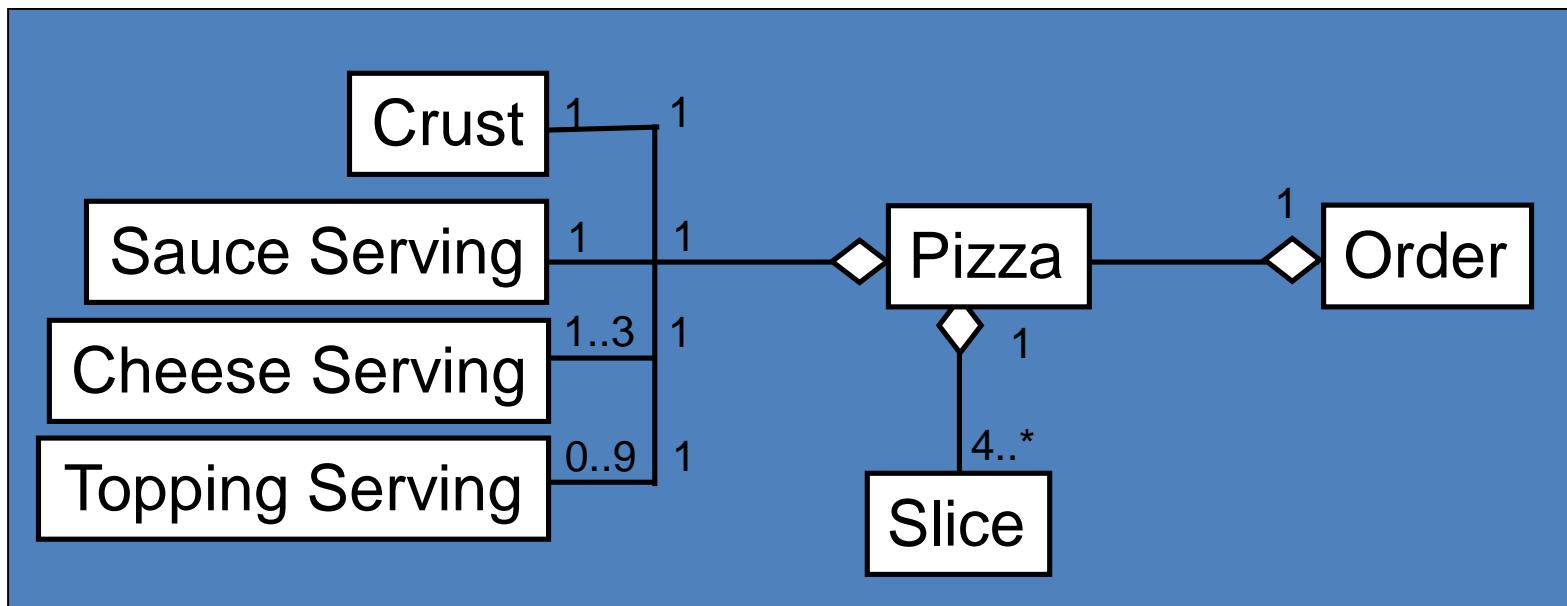
We can model objects that contain other objects by way of special associations called *aggregations* and *compositions*.

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 adornment on the association.



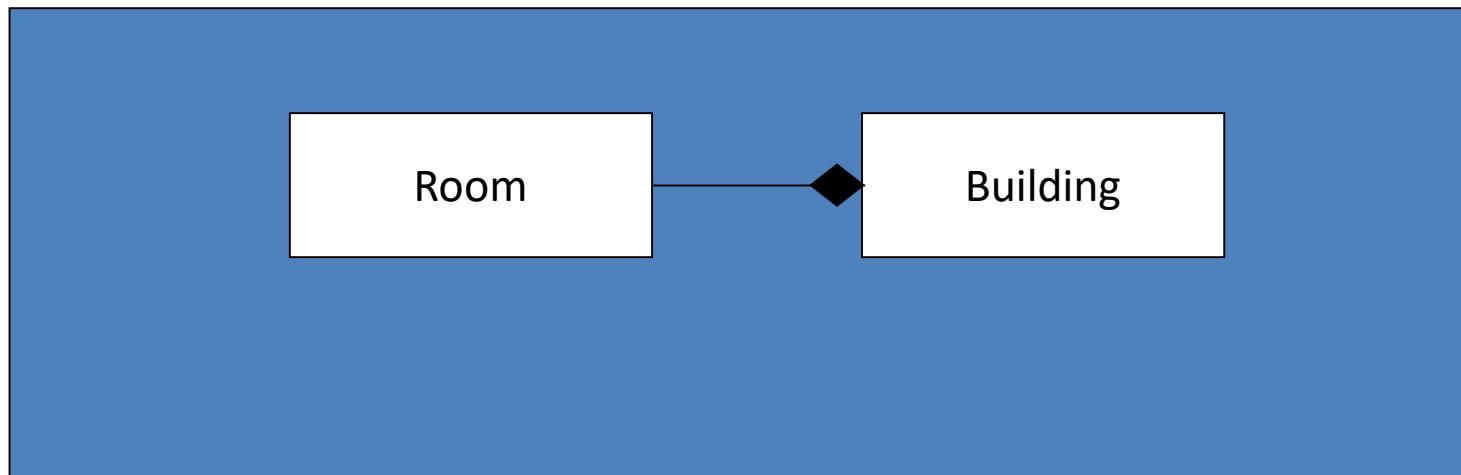
Aggregation

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



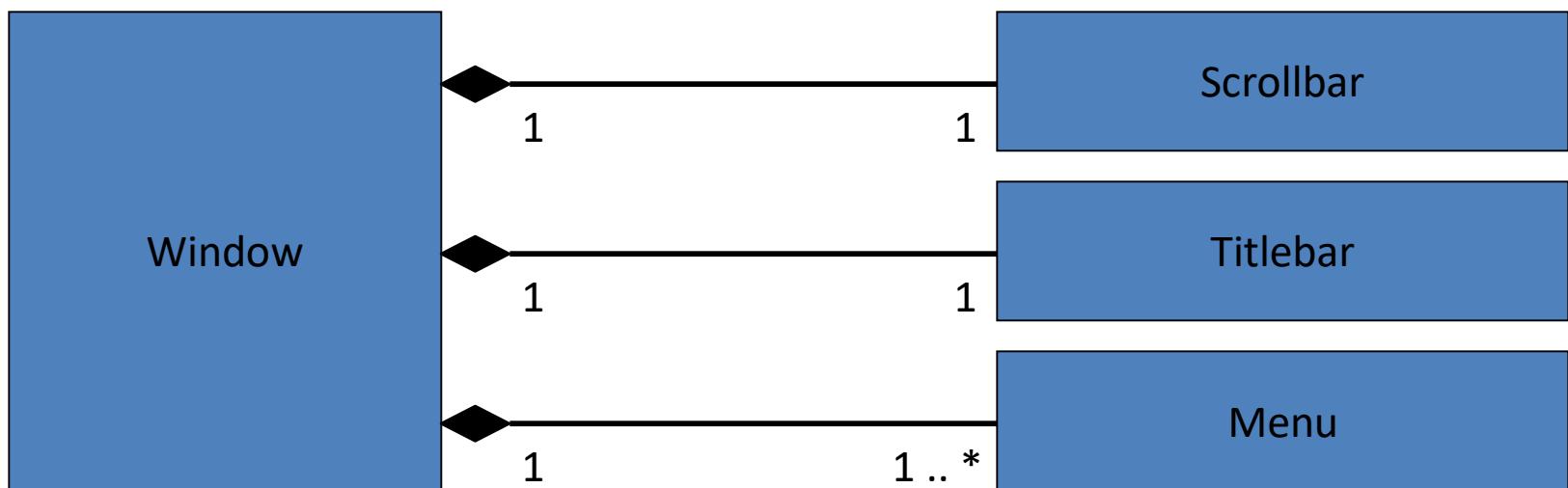
OO Relationships: Composition (very similar to aggregation)

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

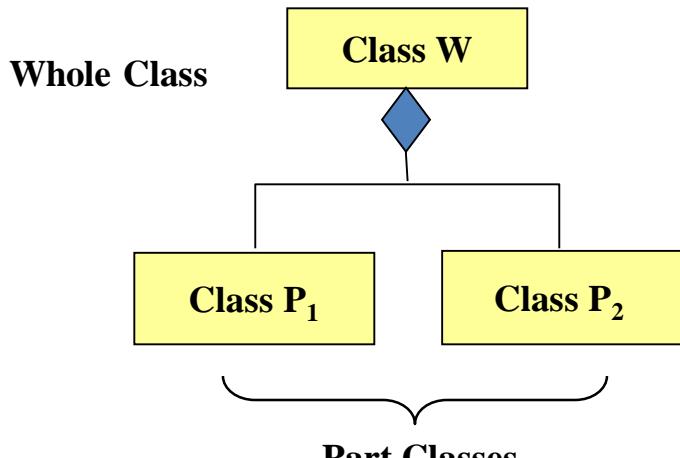


Composition

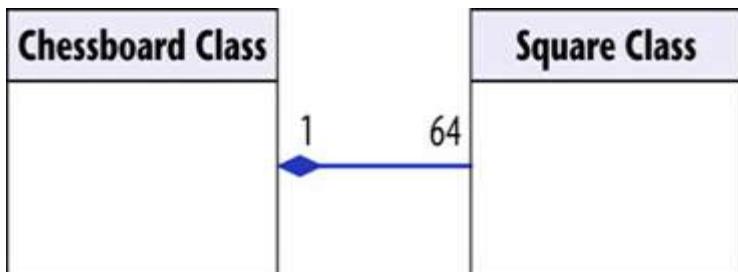
A *composition* indicates a strong ownership and coincident lifetime of parts by the whole (*i.e.*, they live and die as a whole). Compositions are denoted by a filled-diamond adornment on the association.



Composition



Example



Association

Models the part–whole relationship

Composition

Also models the part–whole relationship but, in addition, every part may belong to only one whole, and If the whole is deleted, so are the parts

Example:

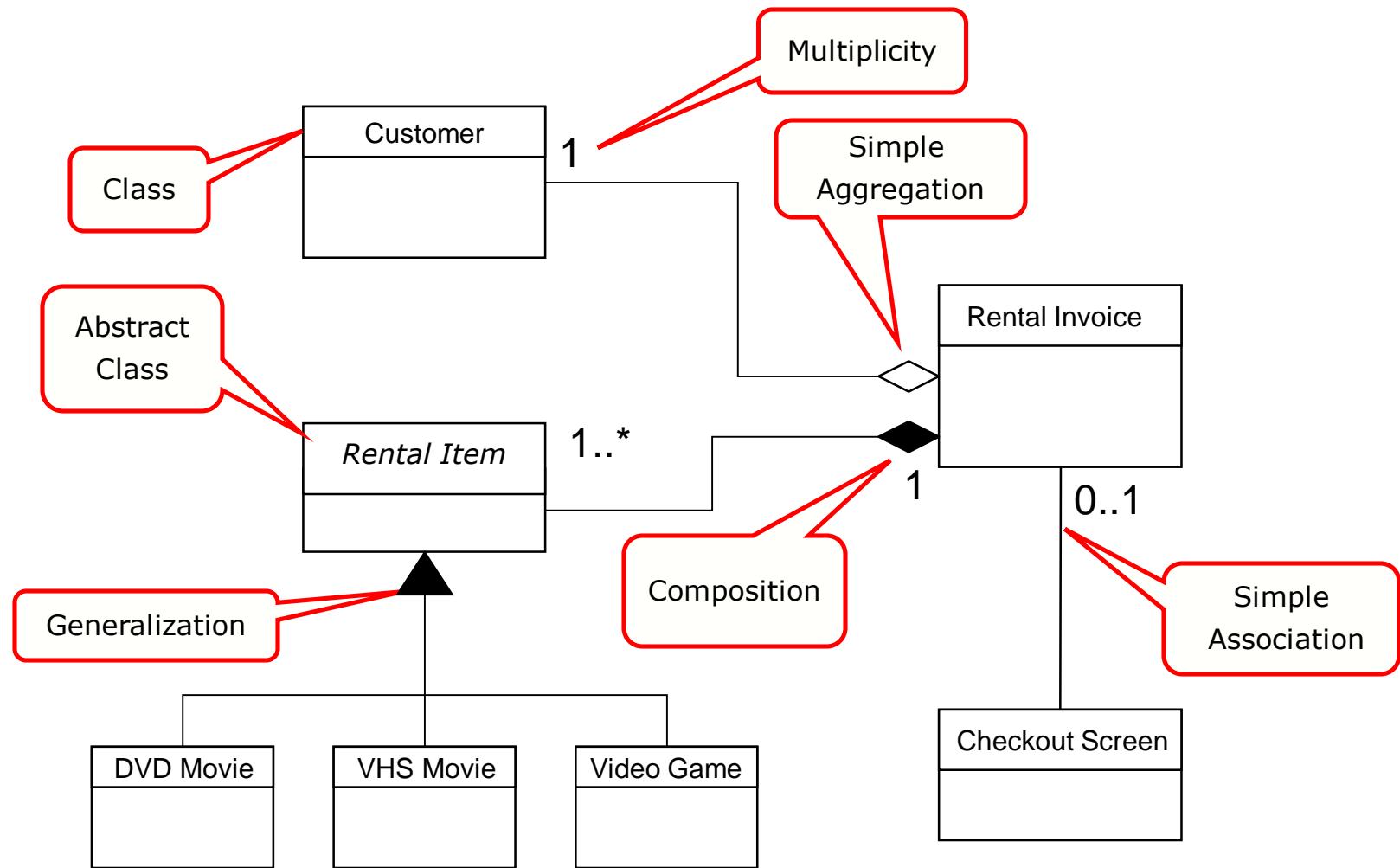
A number of different chess boards: Each square belongs to only one board. If a chess board is thrown away, all 64 squares on that board go as well.

Aggregation vs. Composition

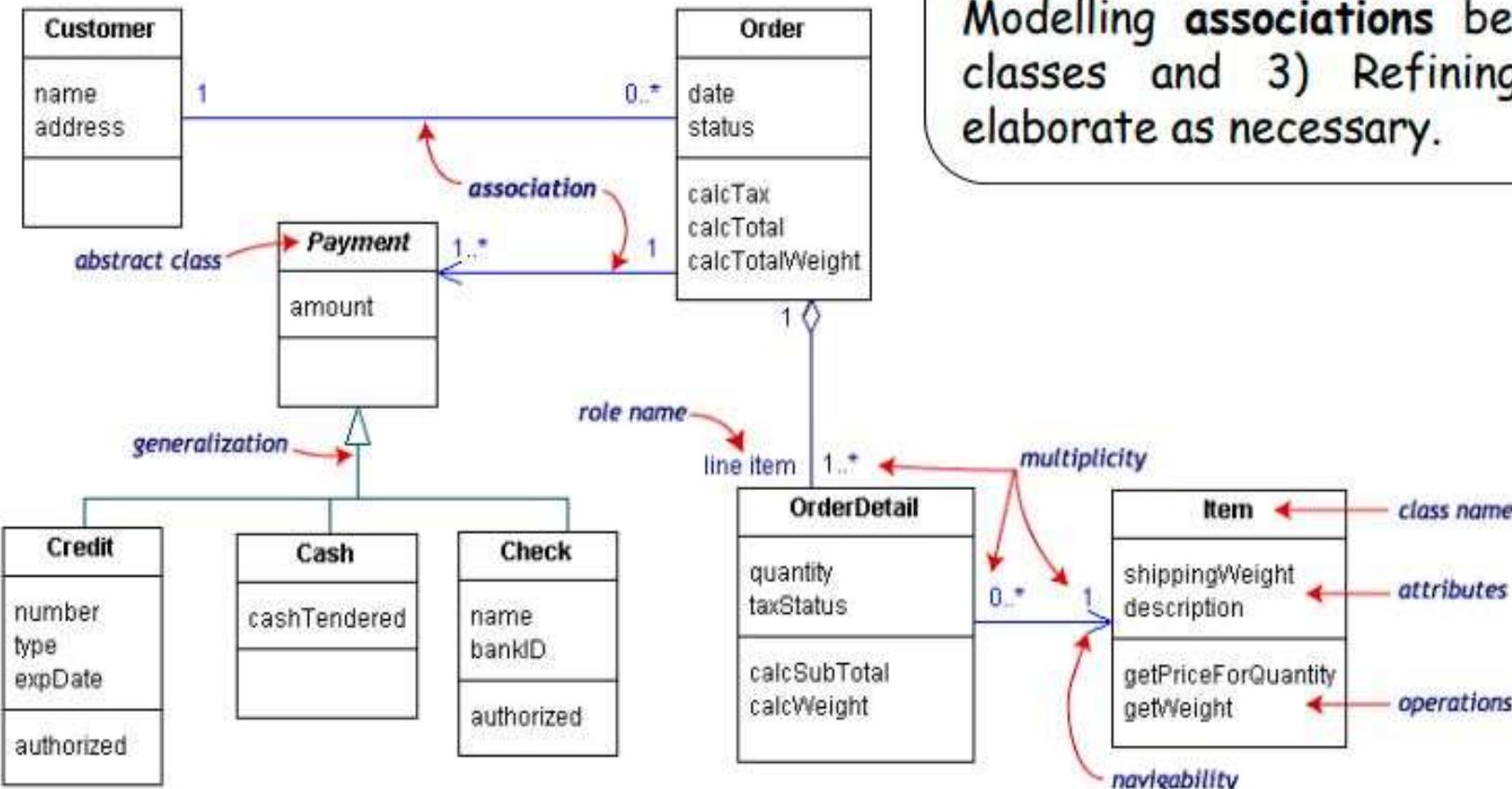
- **Composition** is really a strong form of **association**
 - components have only one owner
 - components cannot exist independent of their owner
 - components live or die with their owner
 - e.g. Each car has an engine that can not be shared with other cars.

- **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

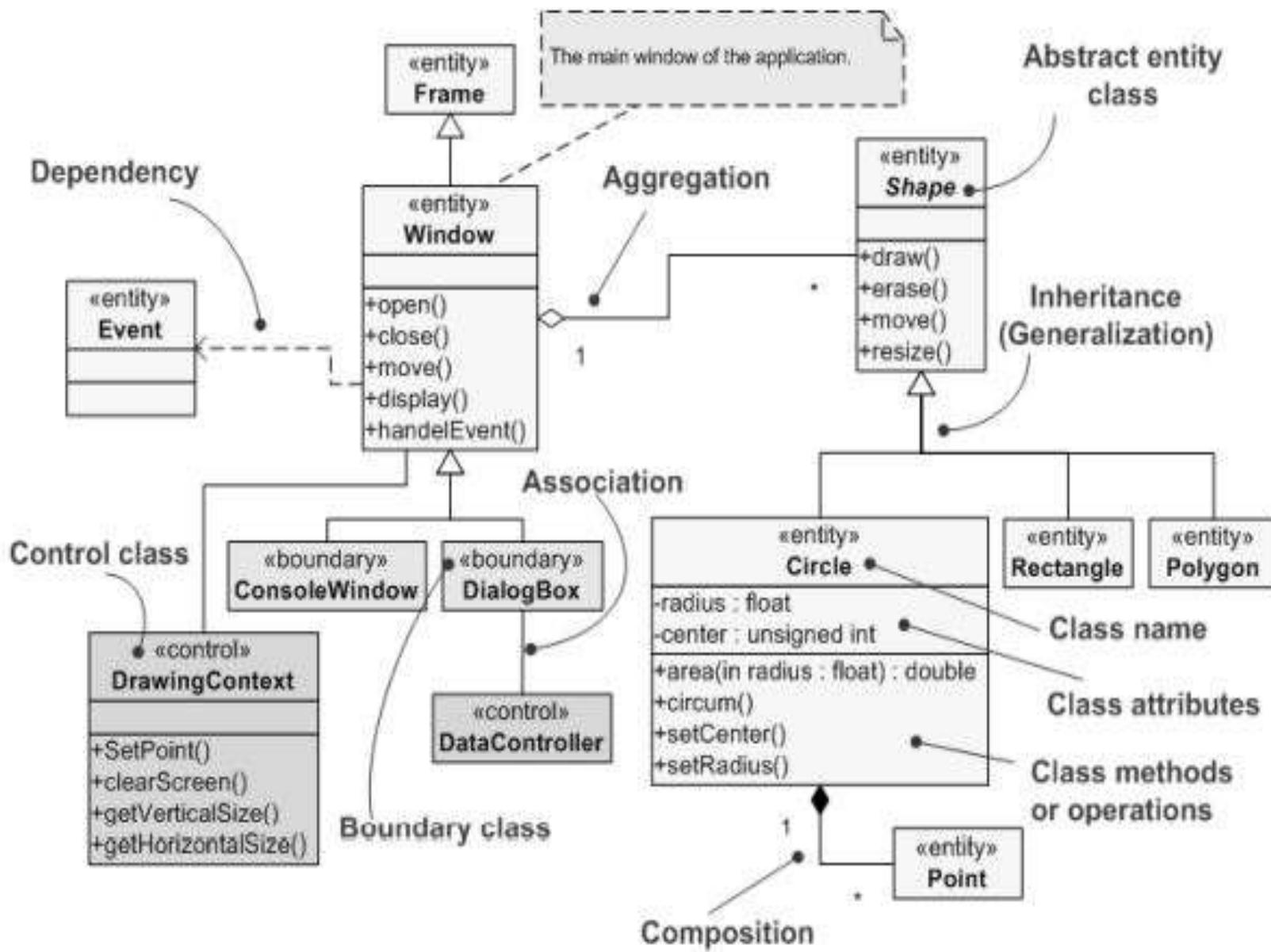


Class diagram example 3



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.

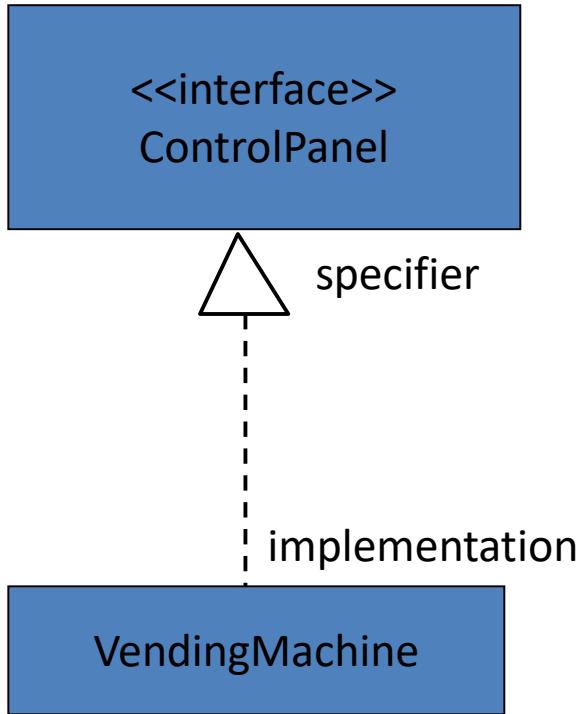
Interface Services

<<interface>>
ControlPanel

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

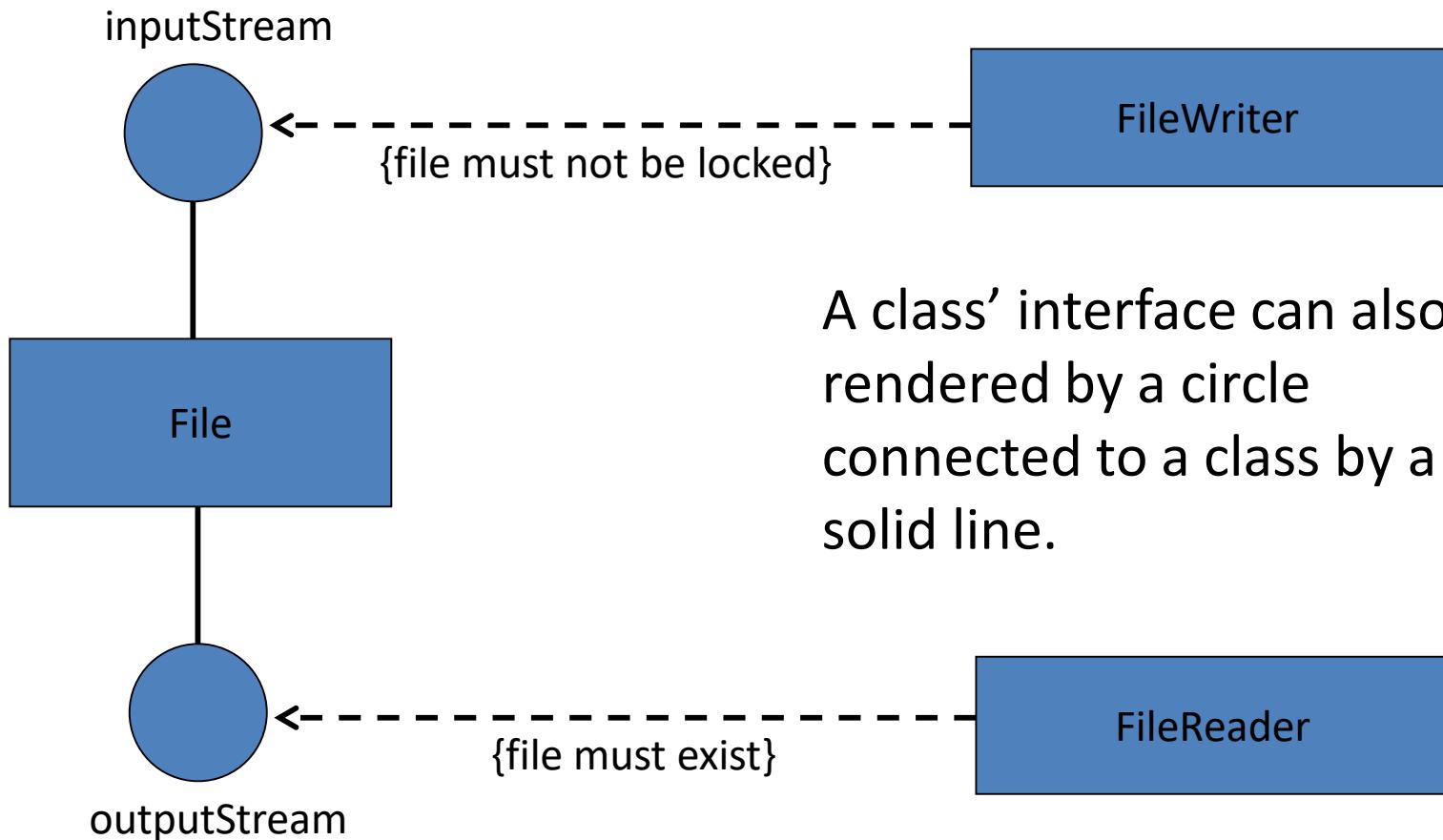
Interfaces do not get instantiated.
They have no attributes or state.
Rather, they specify the services
offered by a related class.

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.

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/>

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

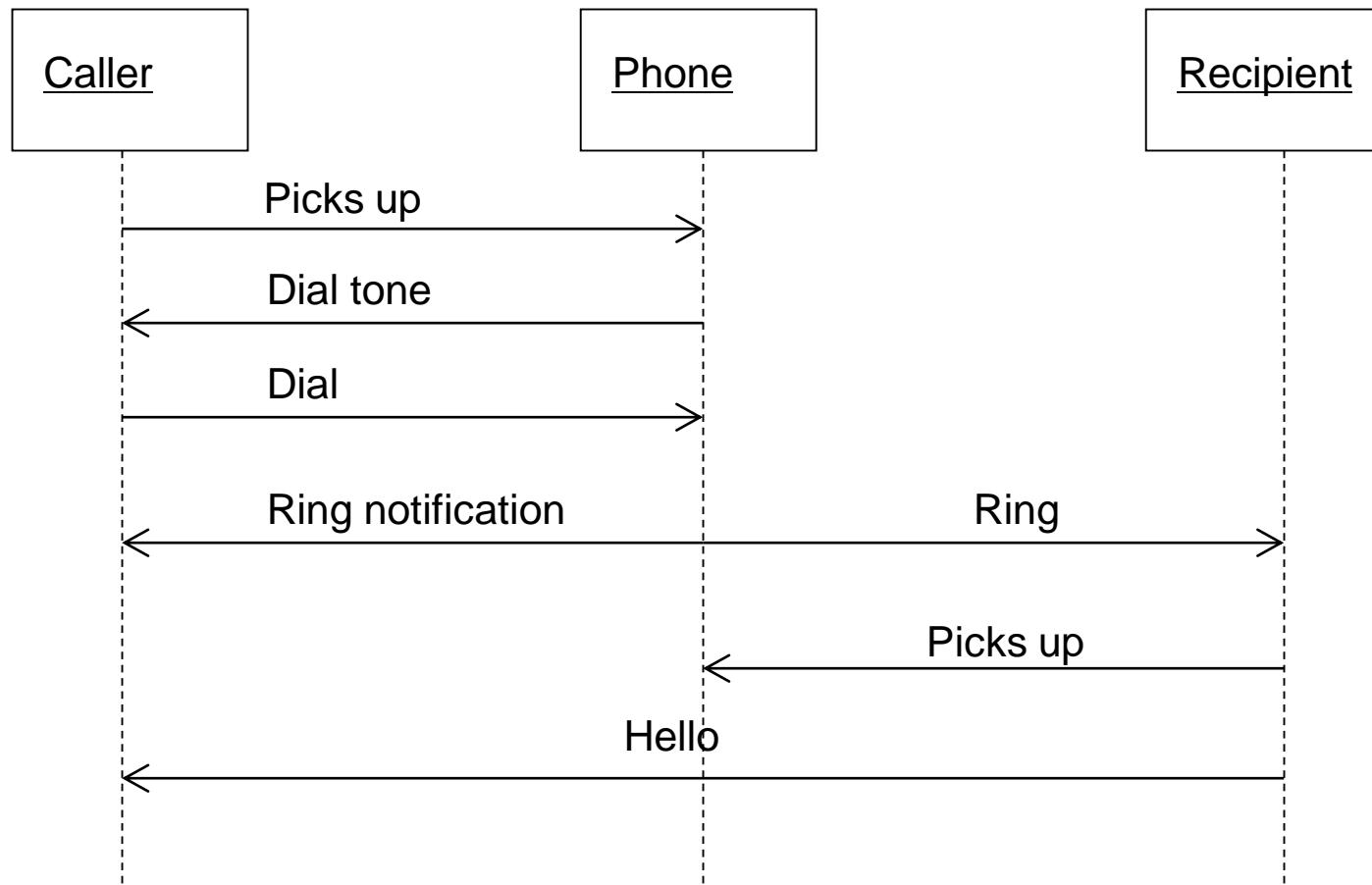
- **Interaction Diagrams** - Set of objects or roles and the messages that can be passed among them.
 - Sequence Diagrams - emphasize time ordering
 - Communication (Collaboration) Diagrams - emphasize structural ordering
- **State Diagrams**
 - State machine consisting of states, transitions, events 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

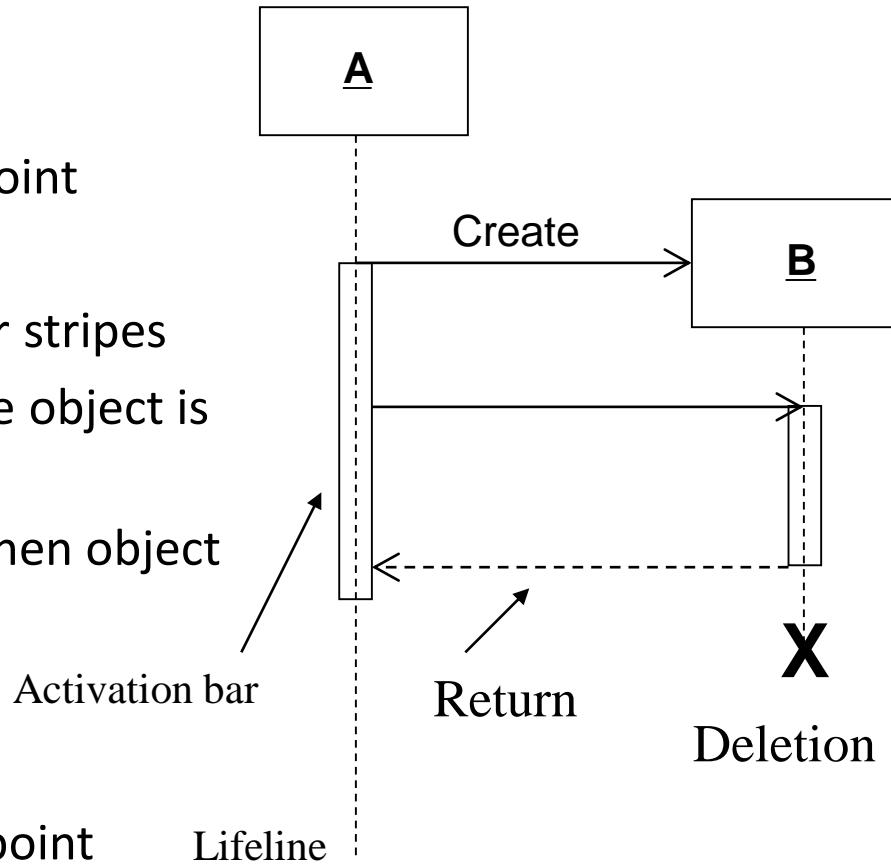
Emphasis on time ordering!

Sequence Diagram(make a phone call)

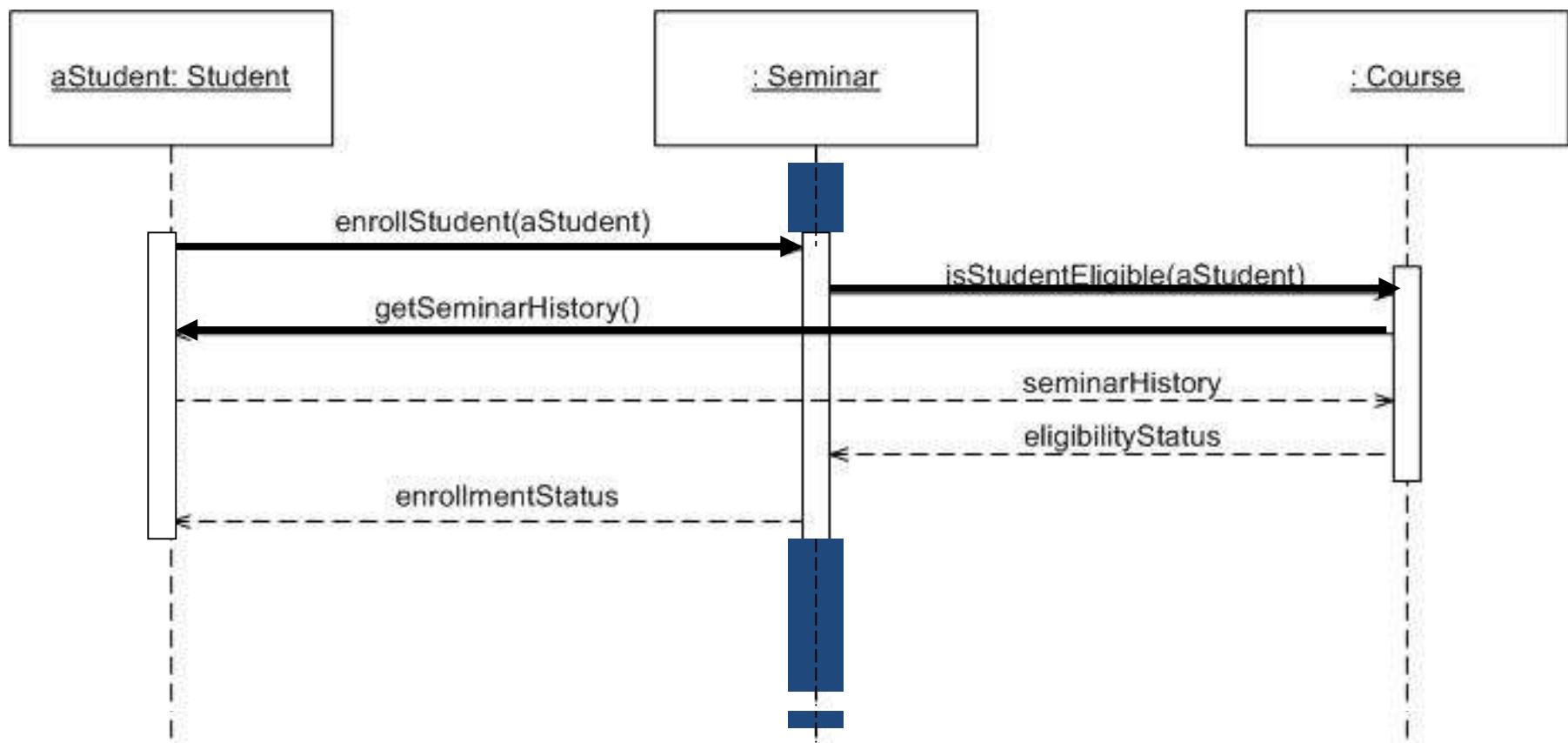


Sequence Diagrams – Object Life Spans

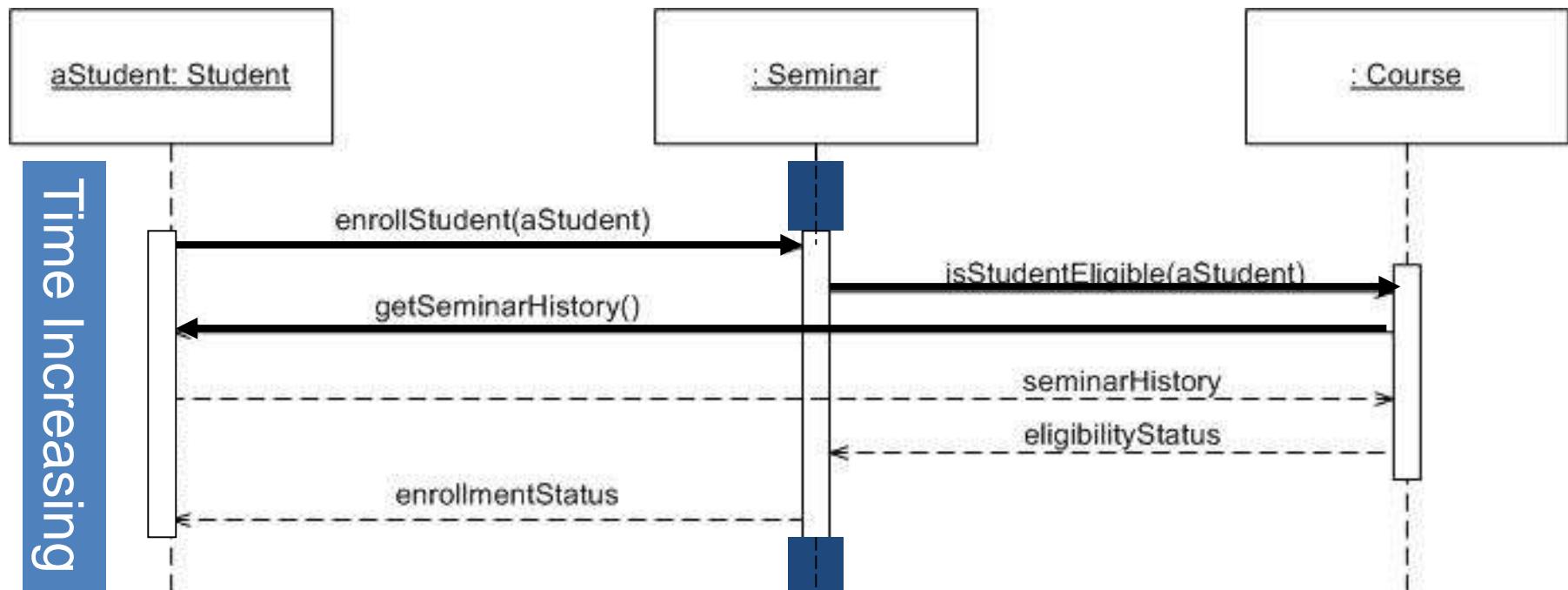
- Creation
 - Create message
 - Object life starts at that point
- Activation
 - Symbolized by rectangular stripes
 - Place on the lifeline where object is activated.
 - Rectangle also denotes when object is deactivated.
- Deletion
 - Placing an 'X' on lifeline
 - Object's life ends at that point



Sequence Diagram



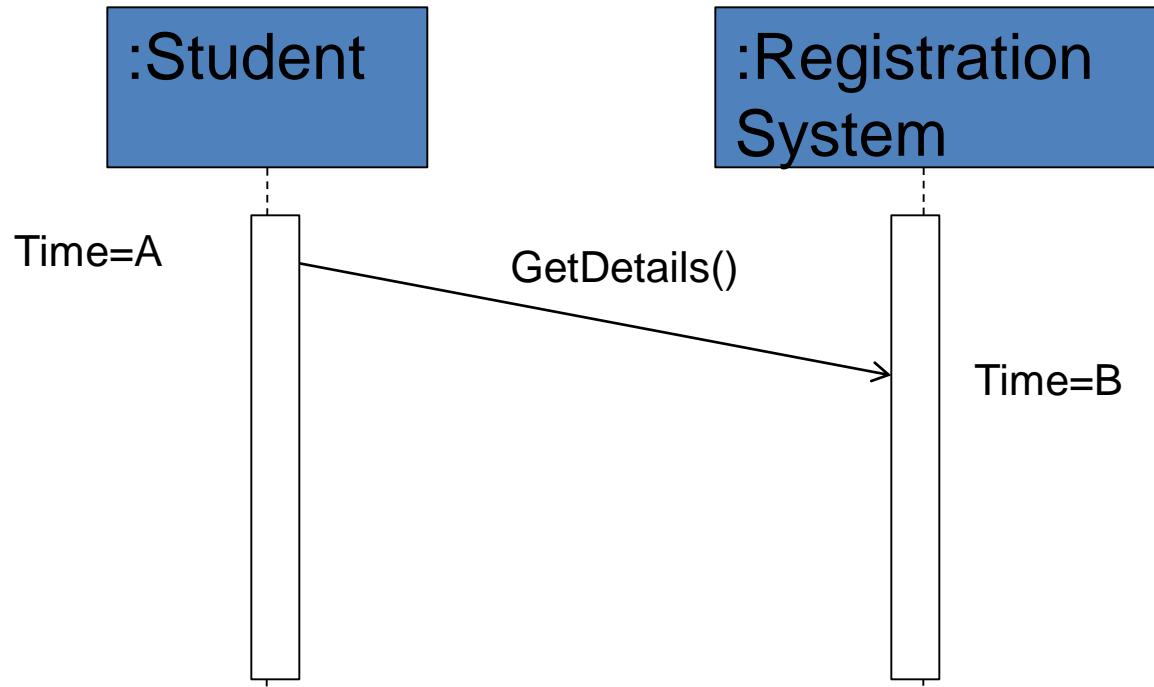
Sequence Diagram



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

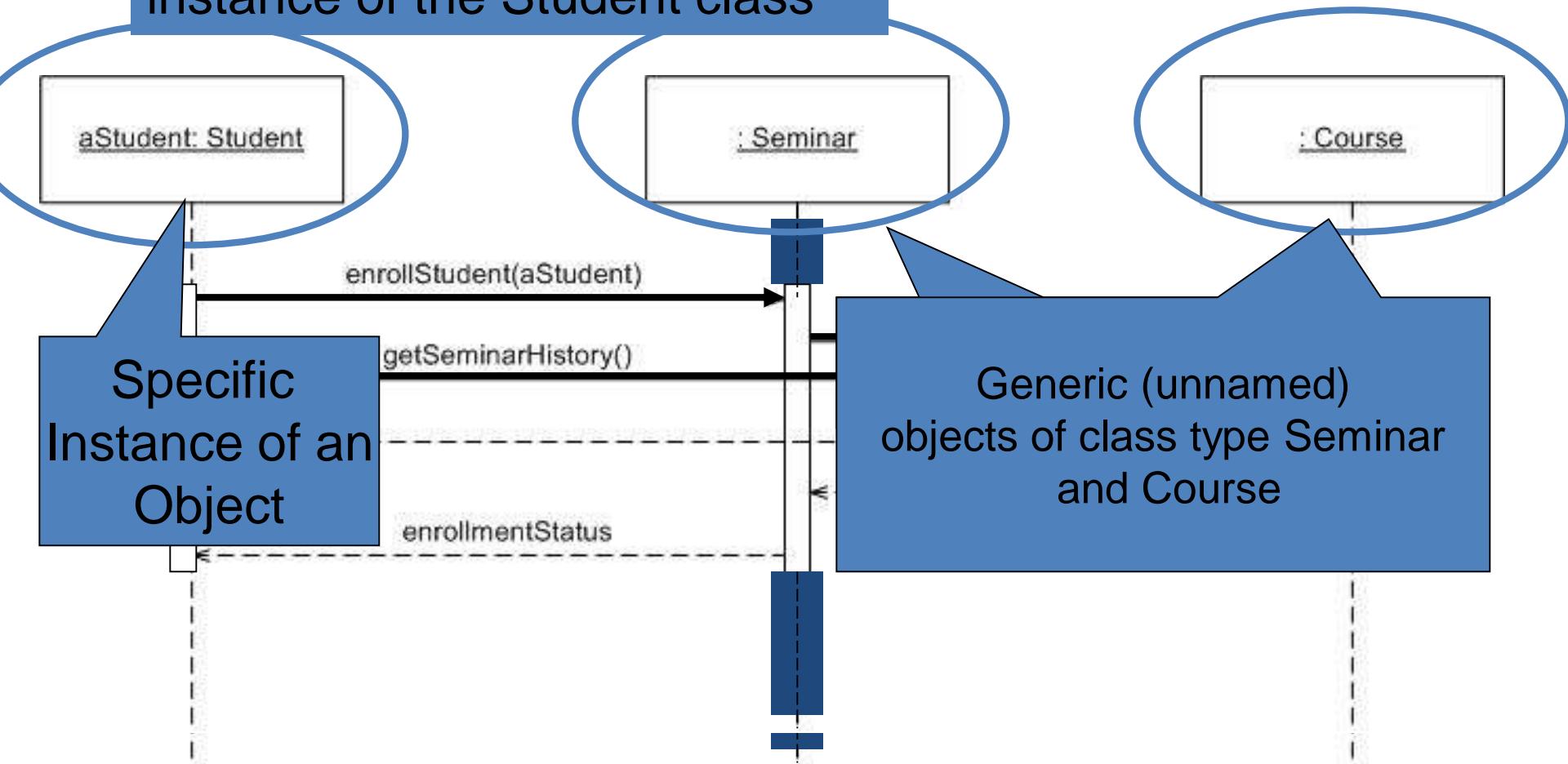


- What does this mean?

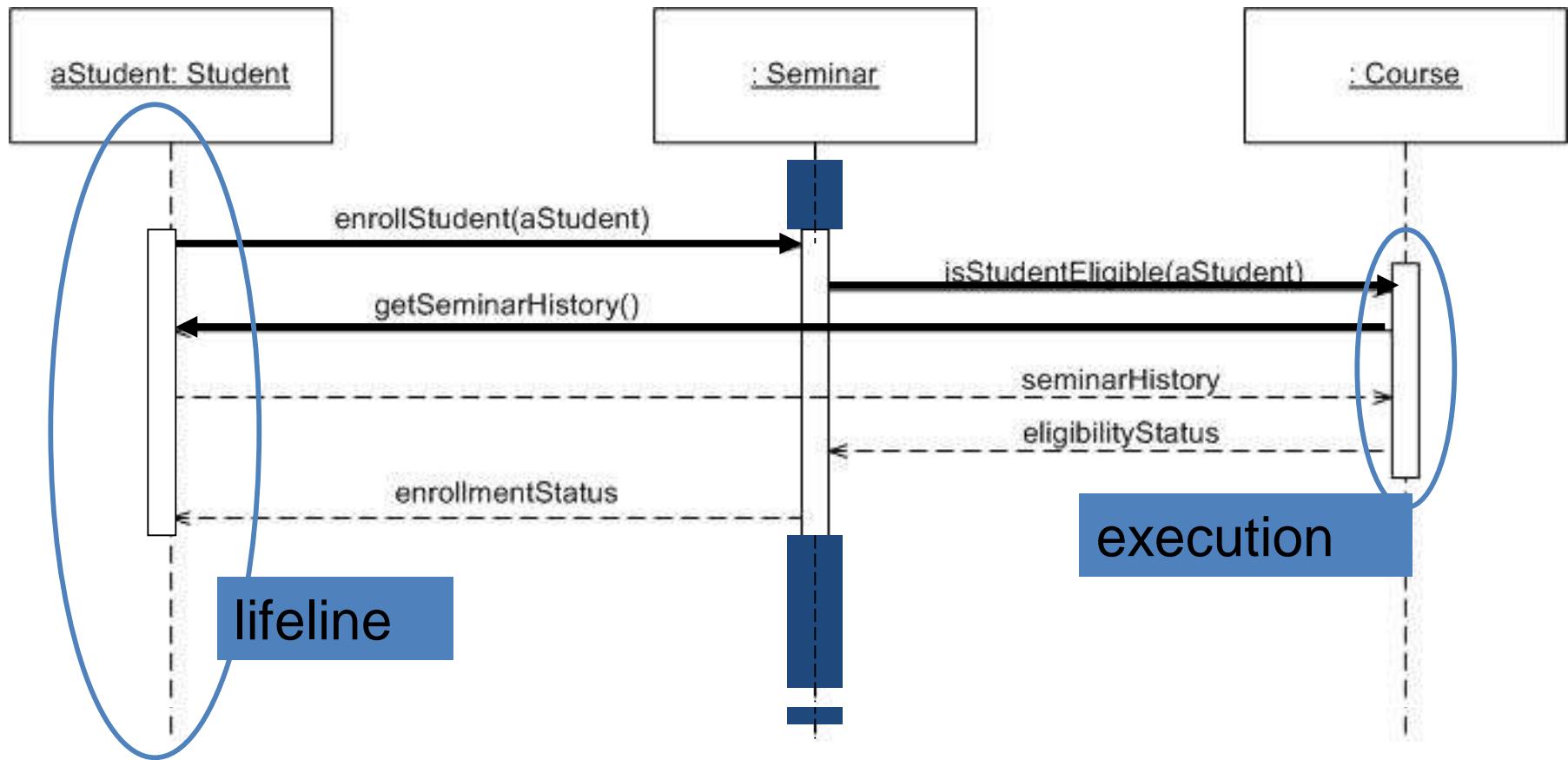
Do you typically care?

Components

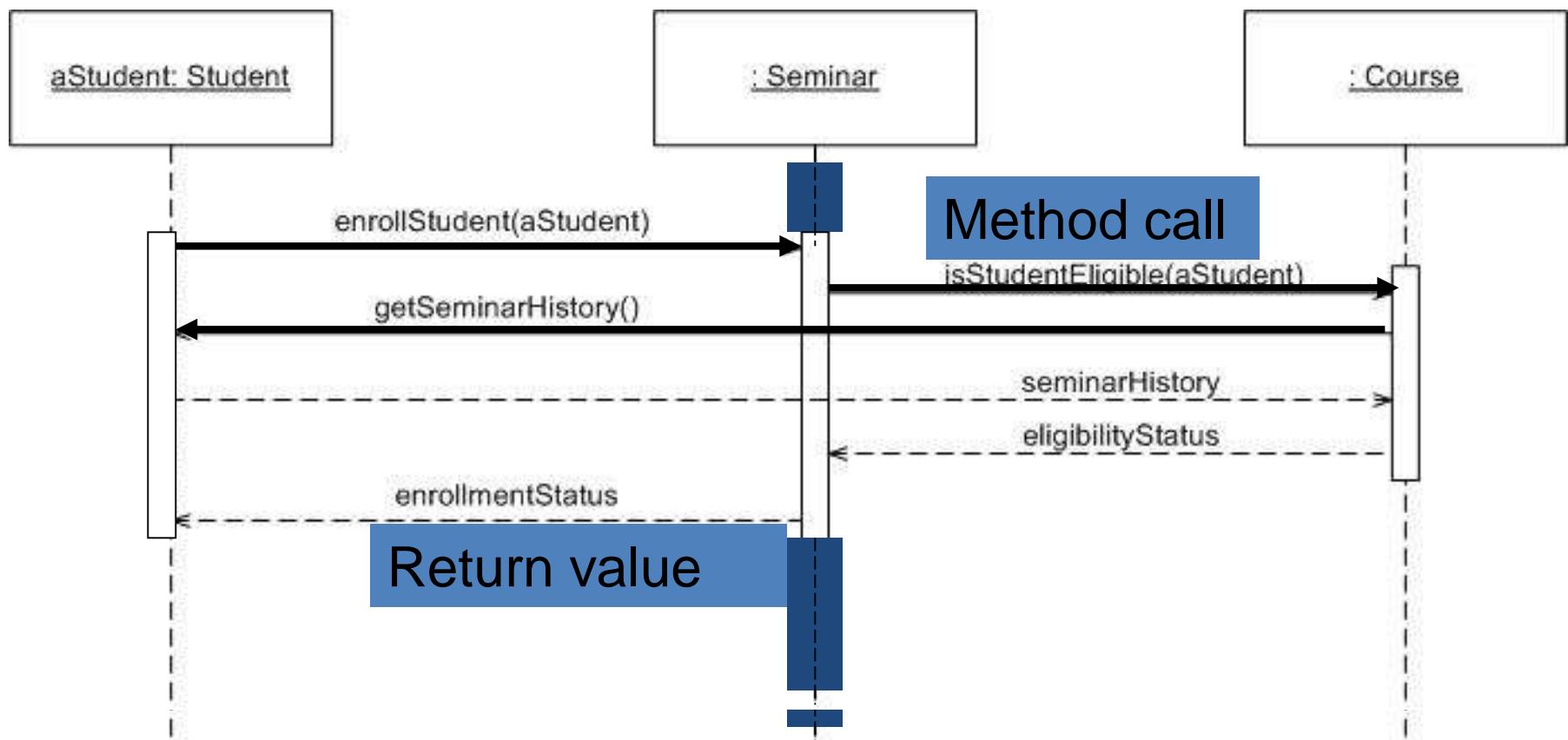
Objects: aStudent is a specific instance of the Student class



Components



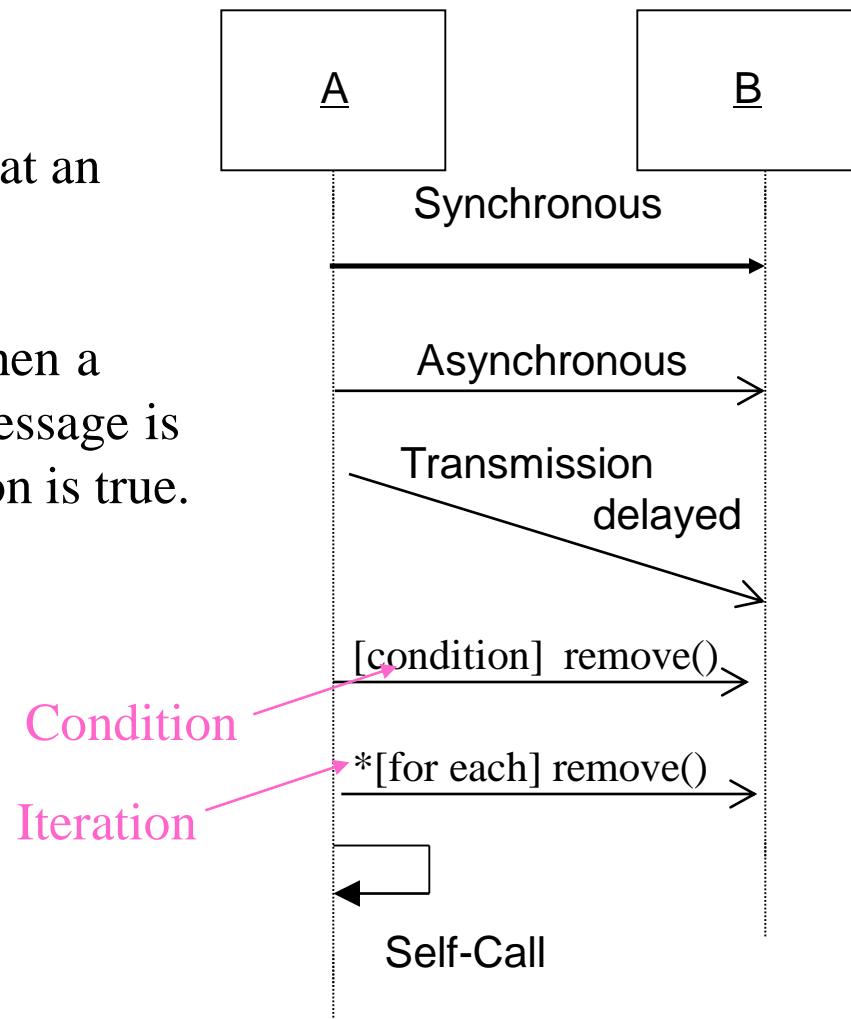
Components



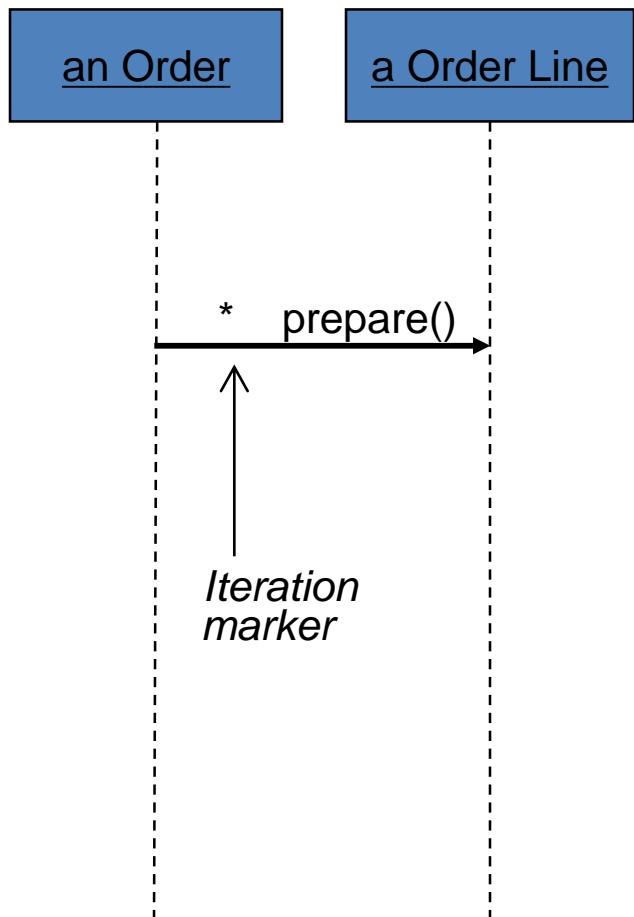
Sequence Diagram: Object interaction

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.

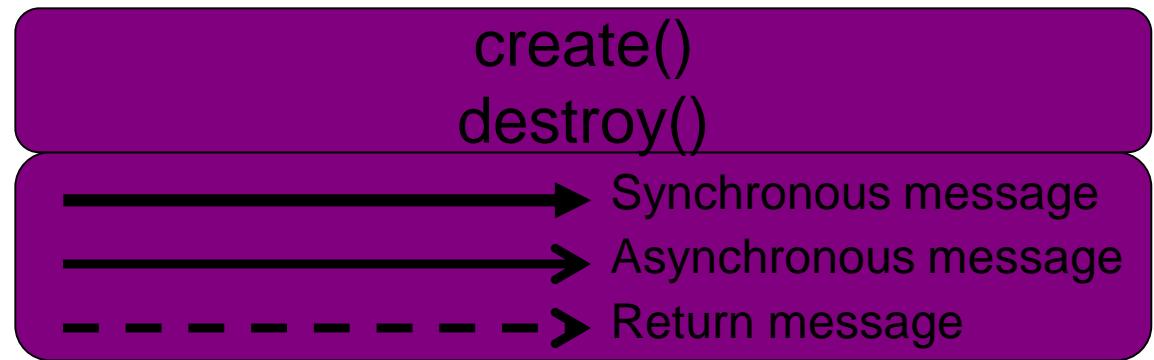
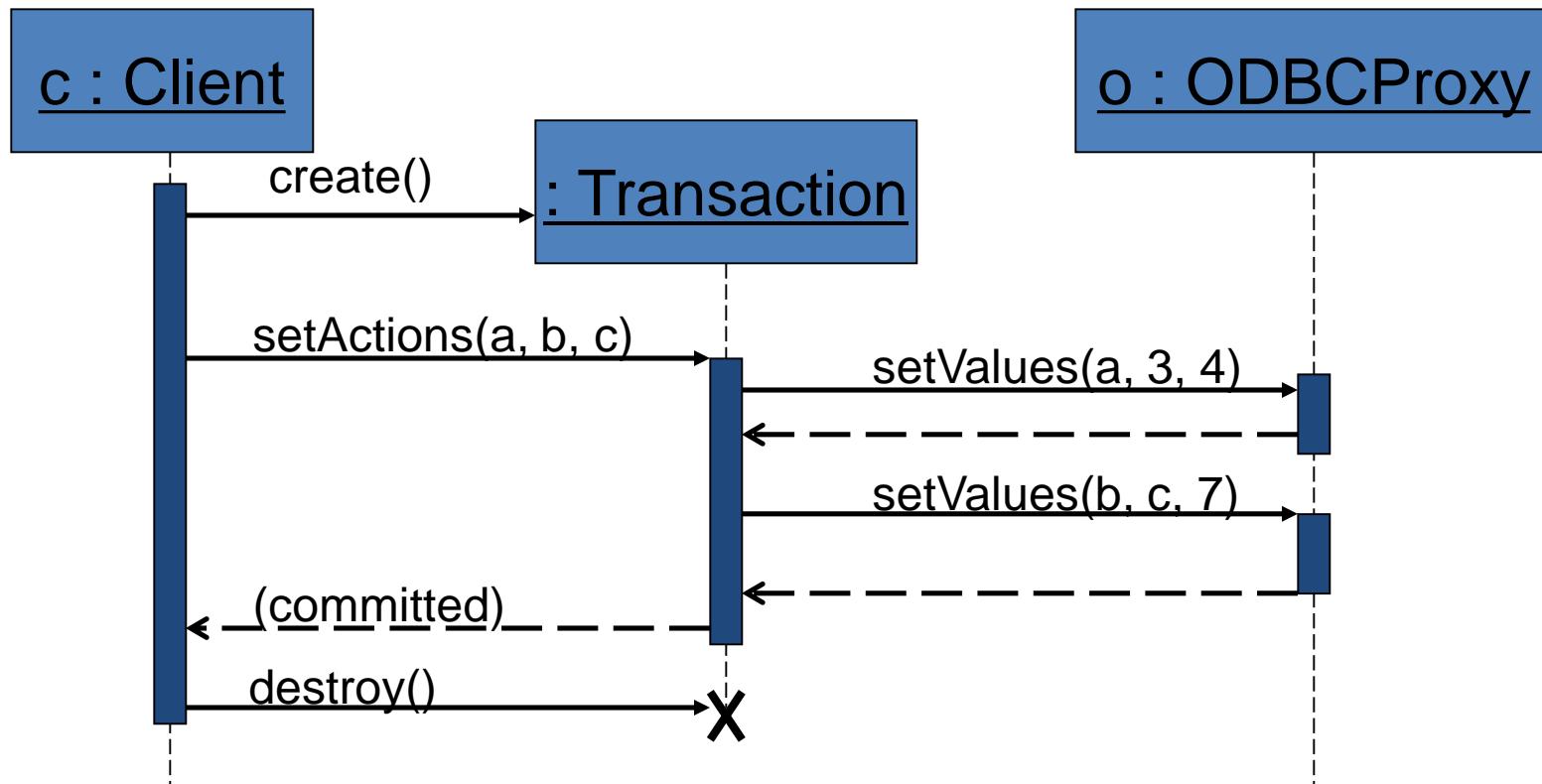


Sequence Diagram

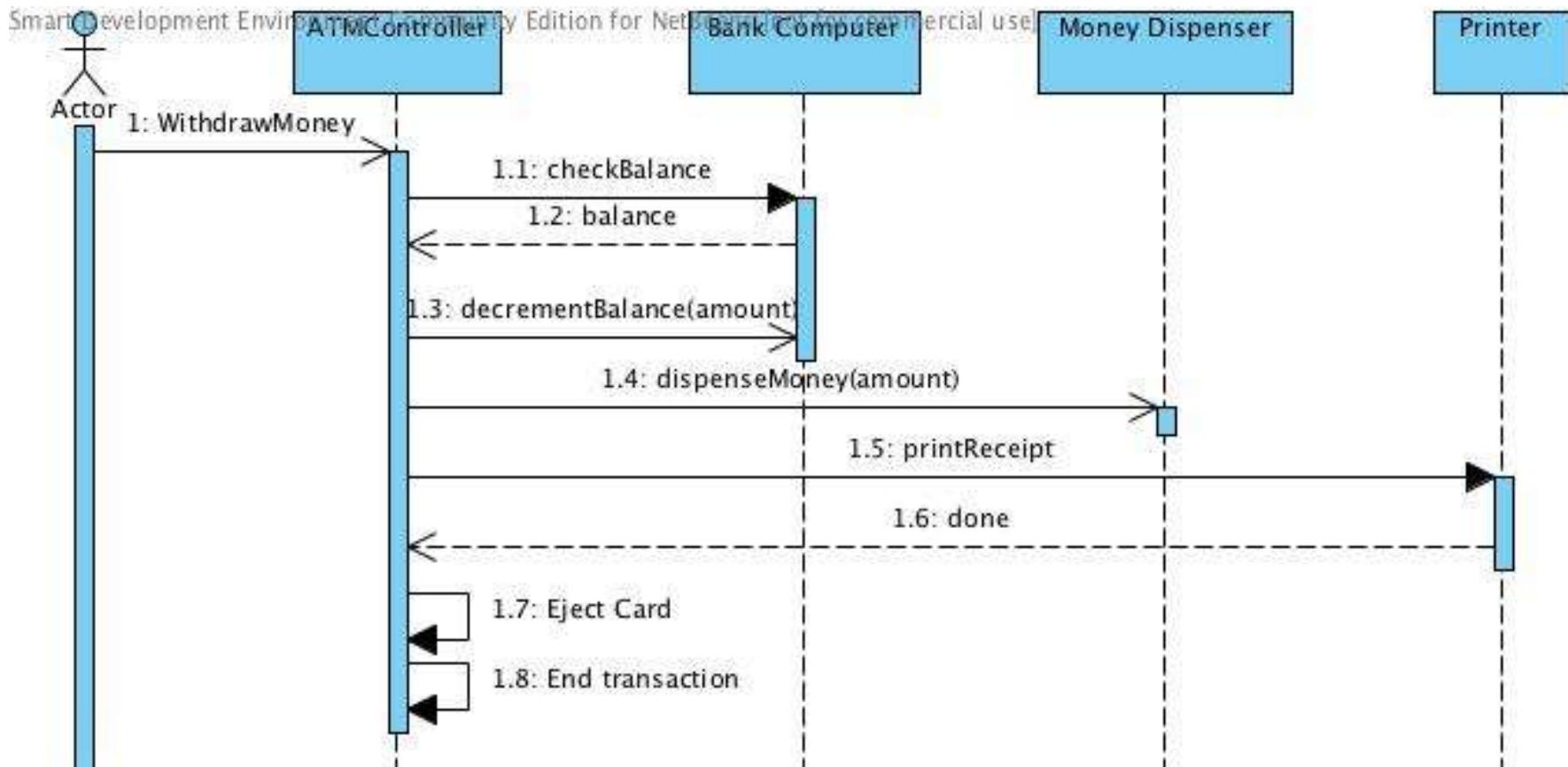


An iteration marker, such as `*` (as shown), or `*[i = 1..n]` , indicates that a message will be repeated as indicated.

Components



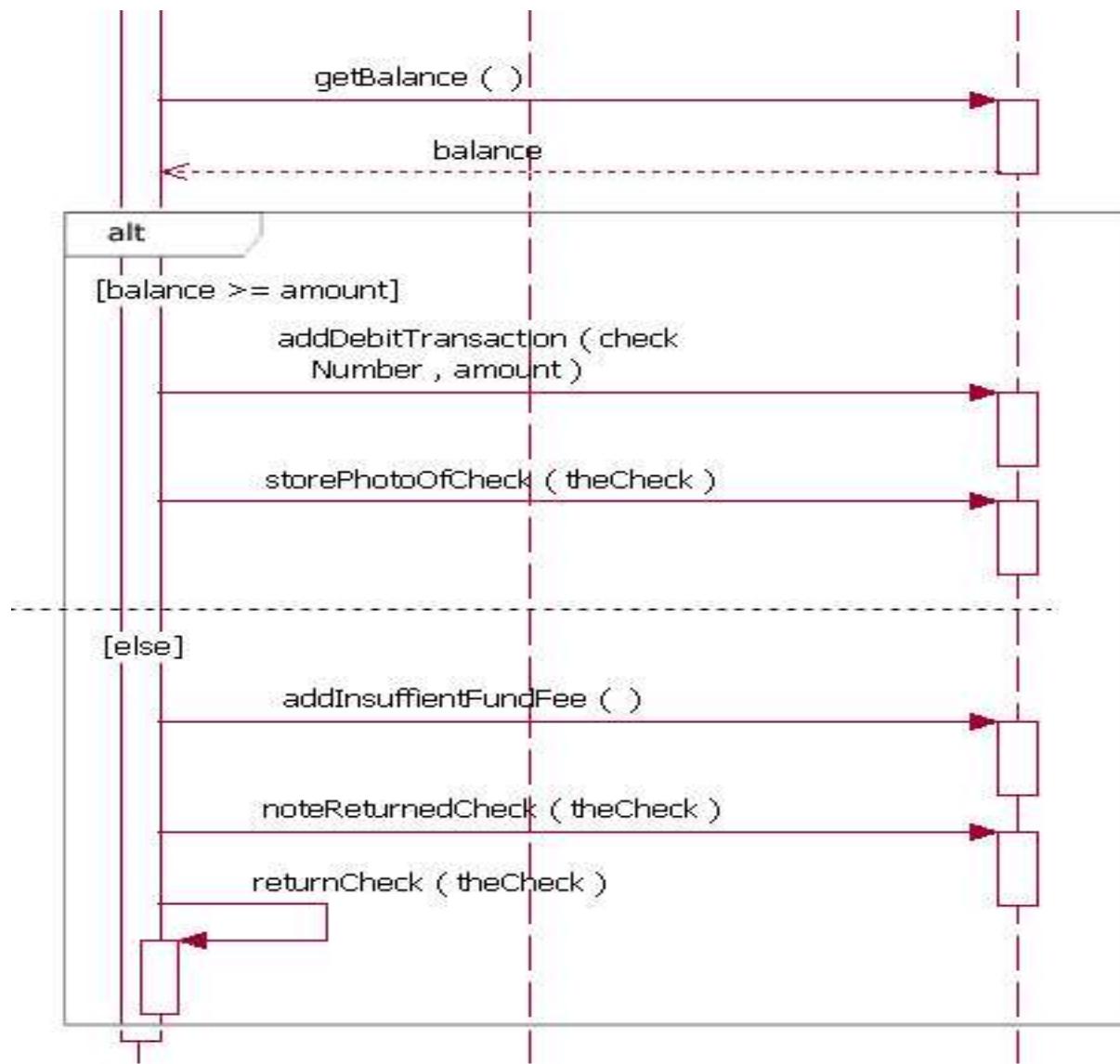
Async Message Example



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

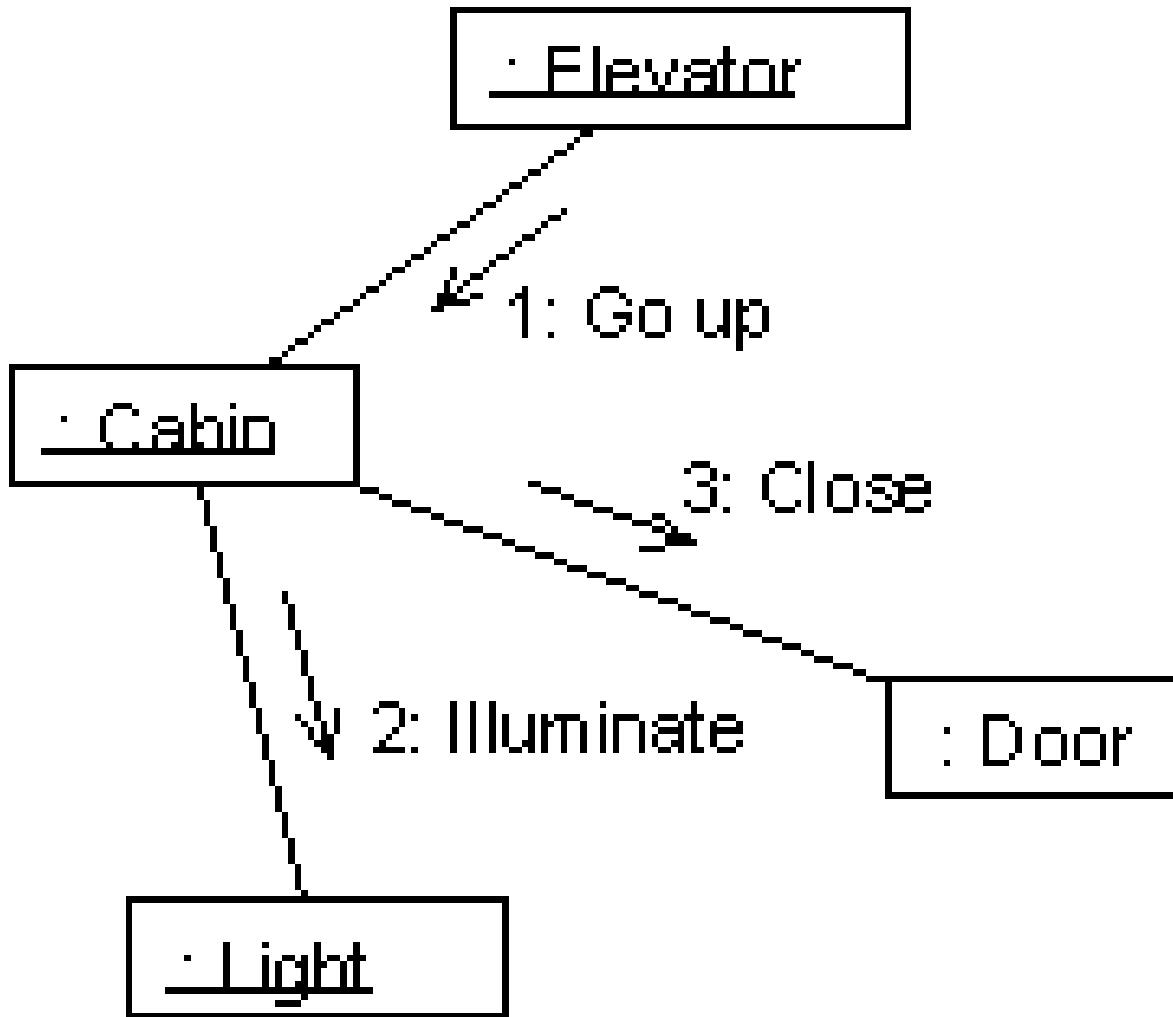
- Sequence diagrams model object interactions with an emphasis on time ordering
- Method call lines
 - Must be horizontal!
 - Vertical height matters! “Lower equals Later”
 - Label the lines
- Lifeline – dotted vertical line
- Execution bar – bar around lifeline when code is running
- Arrows
 - Synchronous 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.

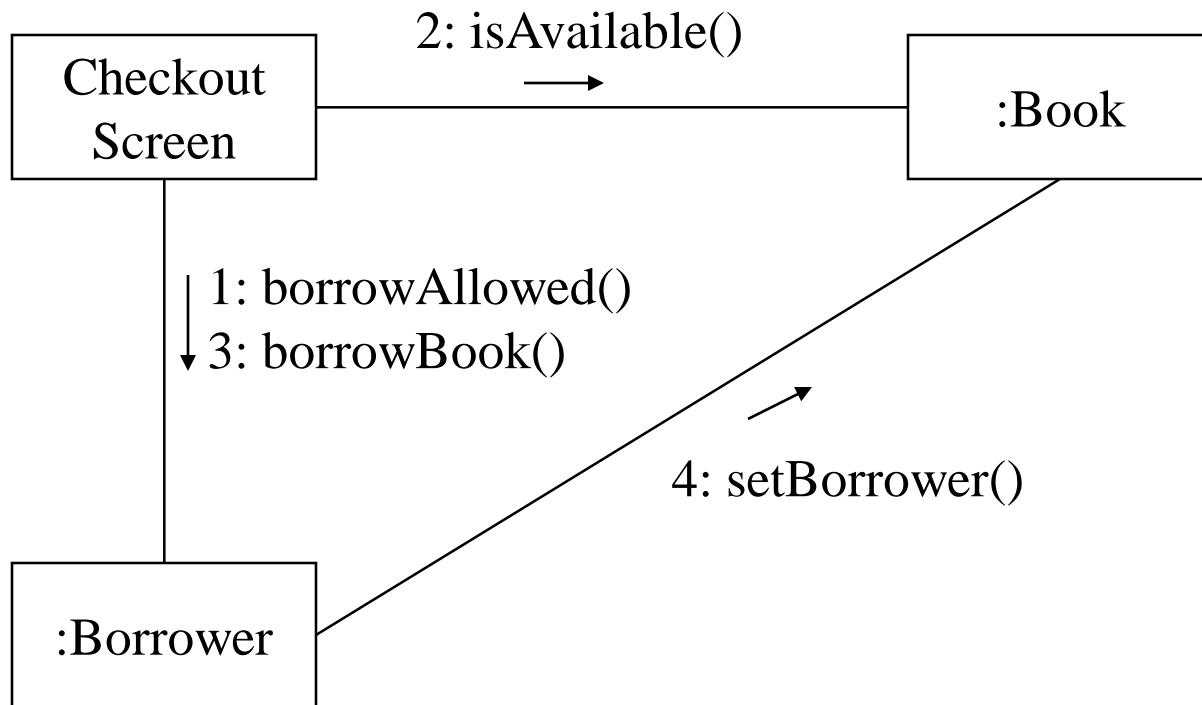
Object identifiers are of the form `objectName : className`, and either the `objectName` or the `className` can be omitted, and the placement of the colon indicates either an `objectName:` , or a `:className`.

Ex. Collaboration Diagram



Example:

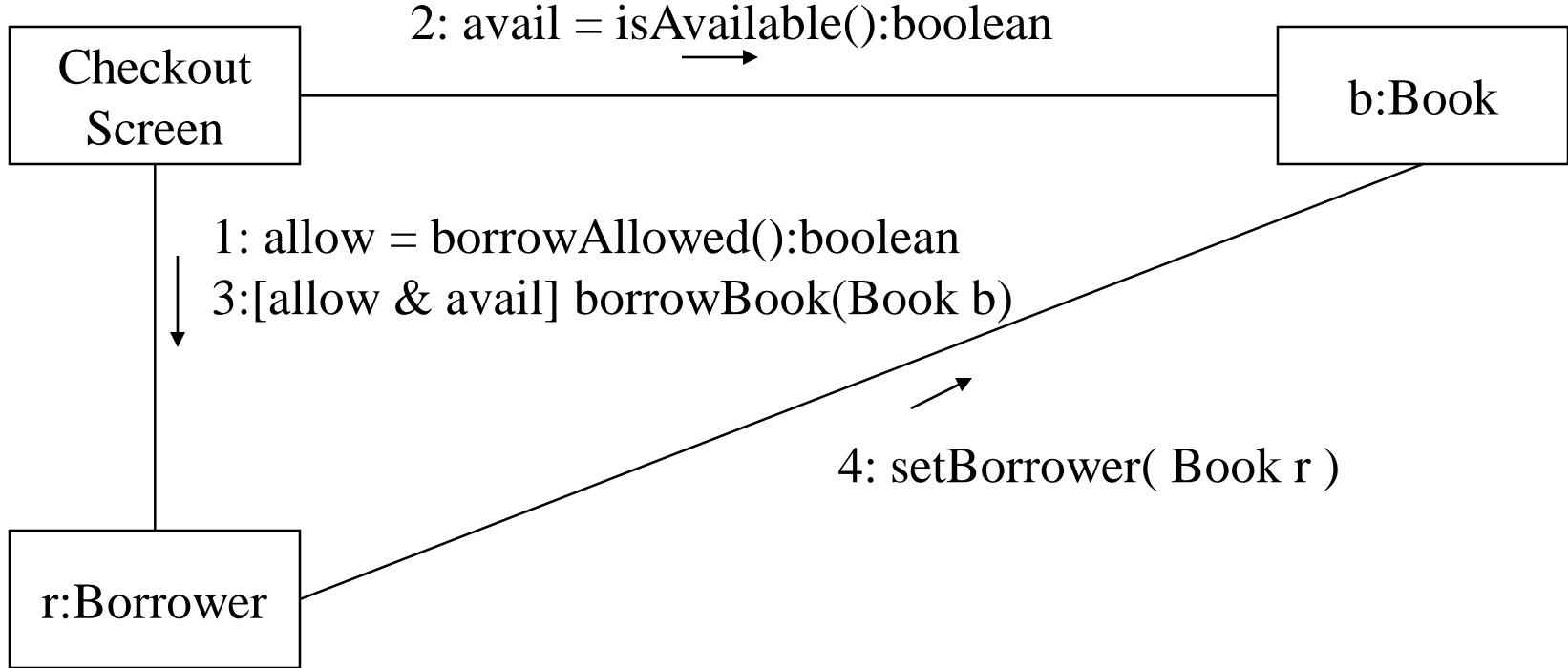
Collaboration Diagram



Interaction (Collaboration) Diagram Notation

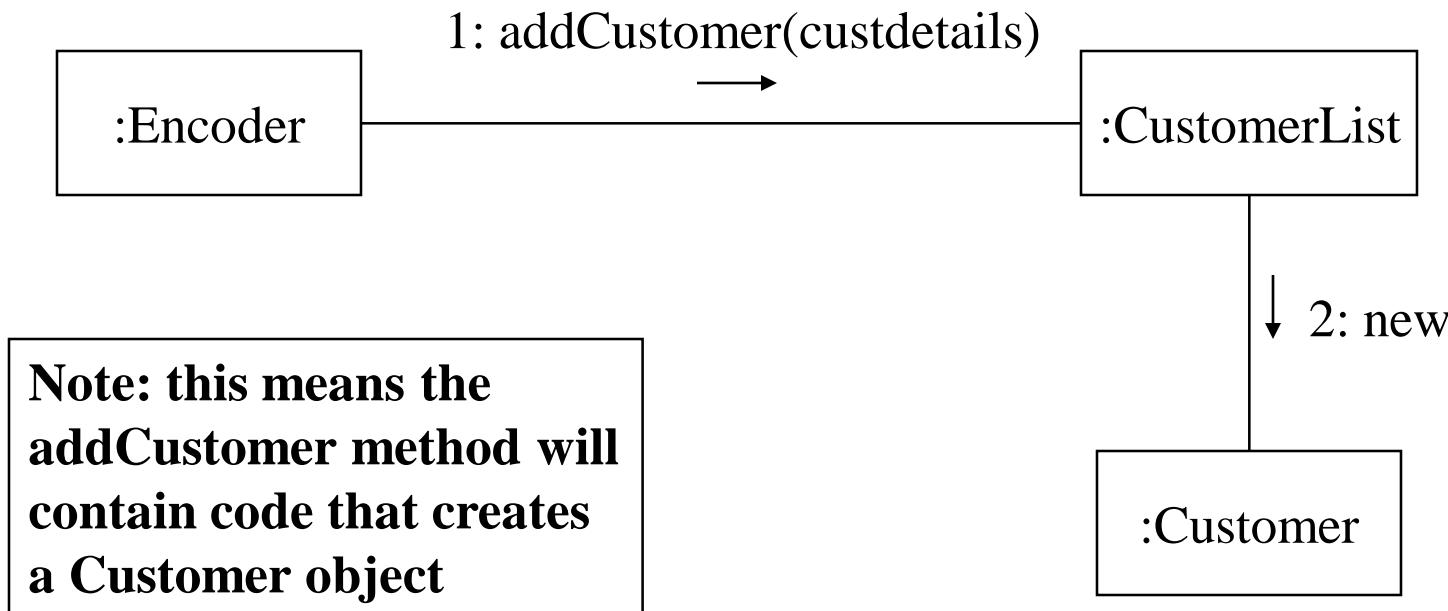
- Rectangles: Classes/Objects
- Arrows: Messages/Method Calls
- Labels on Arrows
 - sequence number (whole numbers or X.X.X notation)
 - method name (the message passed)
 - more details, if helpful and necessary (iterators, conditions, parameters, types, return types)

Including object names, conditions and Types



Creating an Object

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

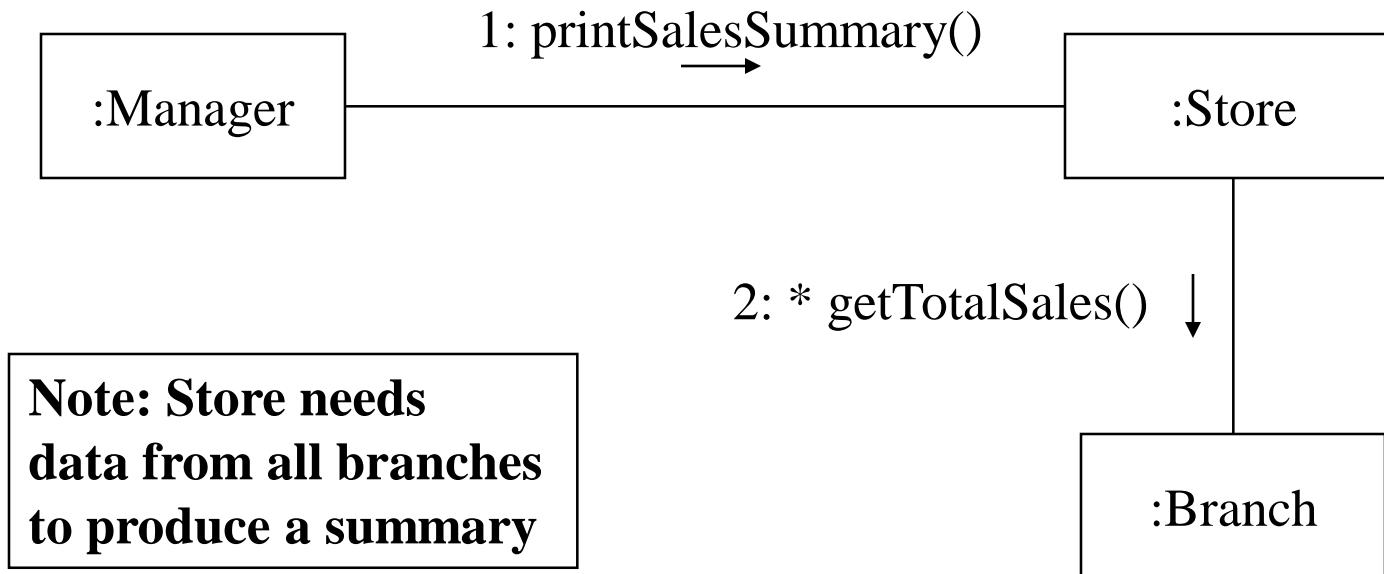


Objects

- 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

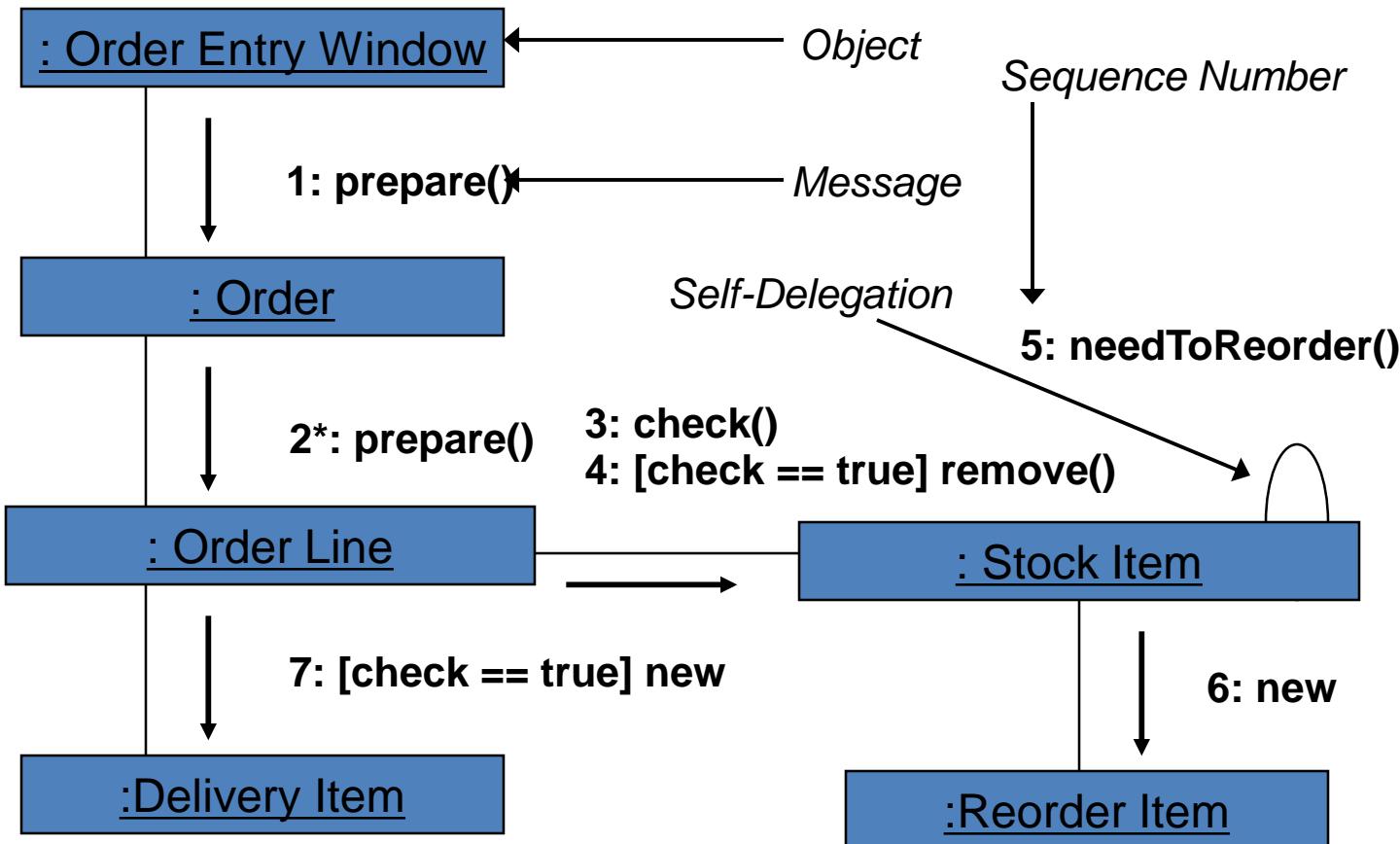
- * is an iterator
 - means the method is called repeatedly



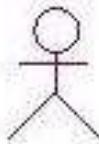
Conditional Messages

- 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



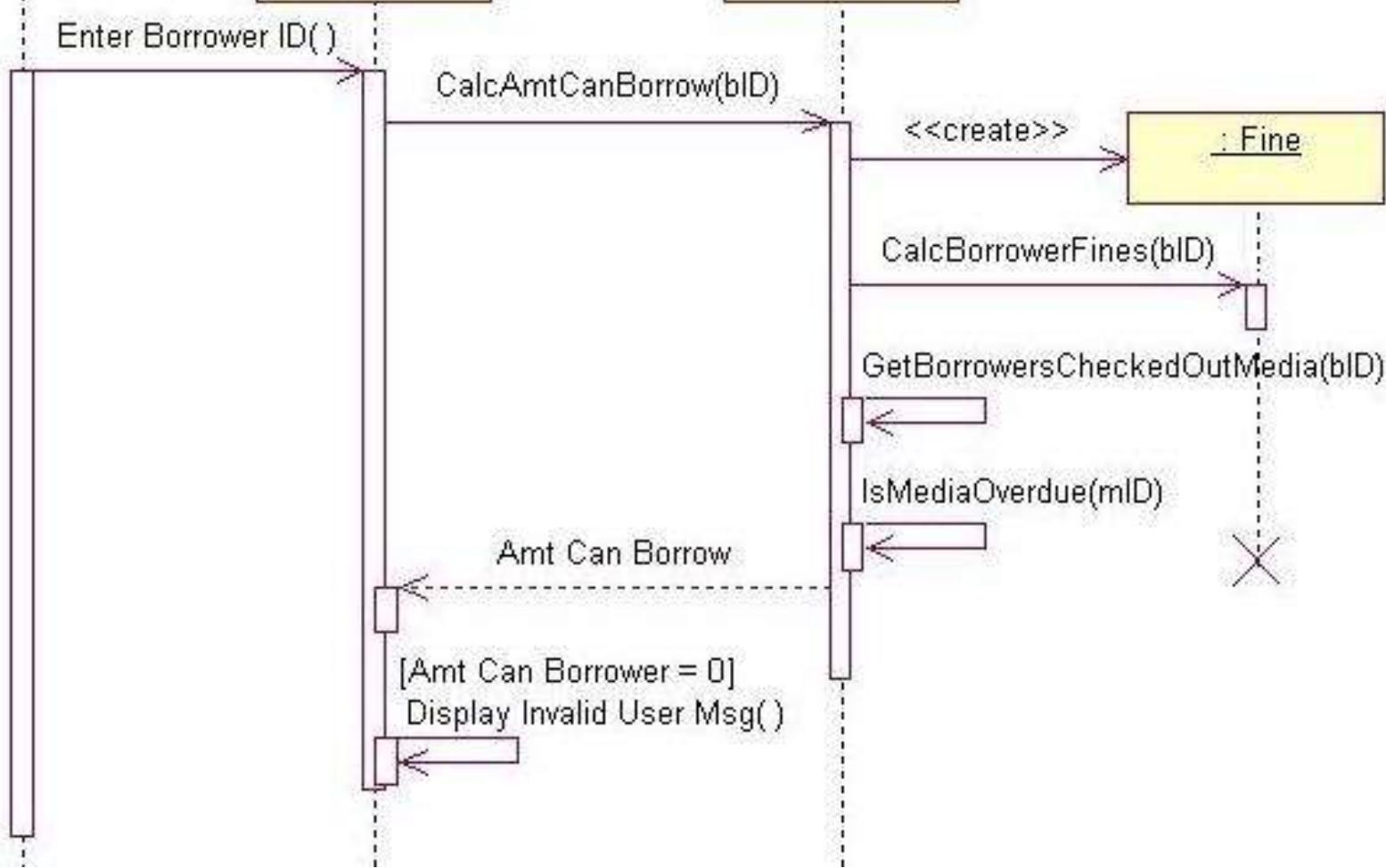
[Fowler,97]



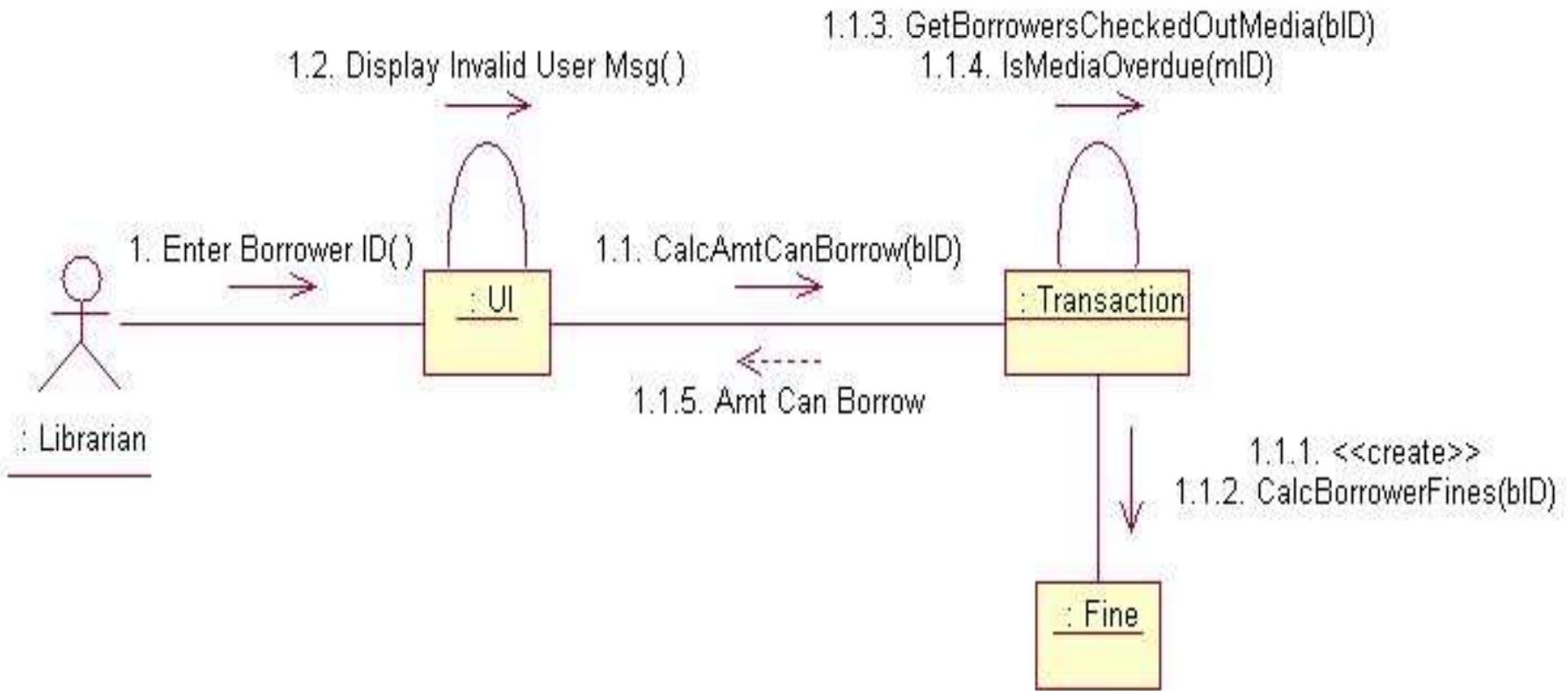
: Librarian

: UI

: Transaction



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>

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

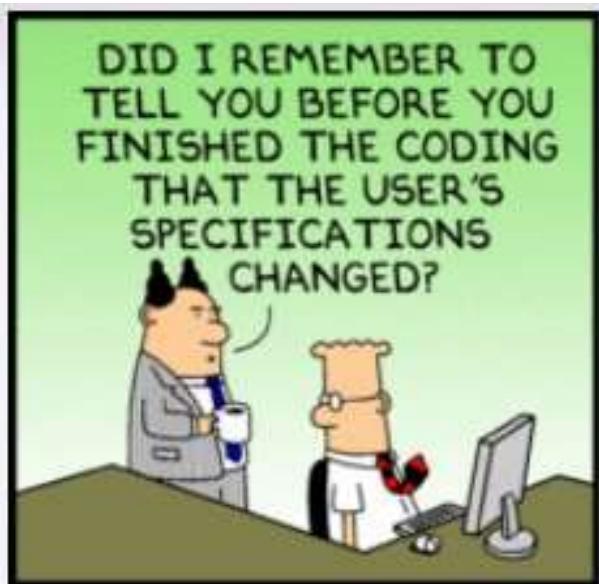
Elaboration

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?

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

WHERE'S
YOUR
ARTIFICIAL
SENSE OF
URGENCY?

TEAM-
WORK
KILLED
IT.

scottadams@aol.com

www.dilbert.com

7-21-06 © 2006 Scott Adams, Inc./Dist. by UFS, Inc.

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

Purpose

- Specifies the software's operational characteristics
- Indicates the software's interfaces with other system elements
- Establishes constraints that the software must meet
- Provides the software designer with a representation of information, function, and behavior
 - This is later translated into architectural, interface, class/data and component-level designs
- Provides the developer and customer with the means to assess quality once the software is built

Overall Objectives

- Three primary objectives
 - To describe what the customer requires
 - To establish a basis for the creation of a software design
 - To define a set of requirements that can be validated once the software is built
- All elements of an analysis model are directly traceable to parts of the design model, and some parts overlap

Analysis Rules of Thumb

- The analysis model should focus on requirements that are visible within the problem or business domain
 - The level of abstraction should be relatively high
- Each element of the analysis model should add to an overall understanding of software requirements and provide insight into the following
 - Information domain, function, and behavior of the system
- The model should delay the consideration of infrastructure and other non-functional models until the design phase
 - First complete the analysis of the problem domain
- The model should minimize coupling throughout the system
 - Reduce the level of interconnectedness among functions and classes
- The model should provide value to all stakeholders
- The model should be kept as simple as can be

Domain Analysis

- Definition
 - The identification, analysis, and specification of common, reusable capabilities within a specific application domain
 - Do this in terms of common objects, classes, subassemblies, and frameworks
- Sources of domain knowledge
 - Technical literature
 - Existing applications
 - Customer surveys and expert advice
 - Current/future requirements
- Outcome of domain analysis
 - Class taxonomies
 - Reuse standards
 - Functional and behavioral models
 - Domain languages

Analysis Modeling Approaches

- Structured analysis
 - Considers data and the processes that transform the data as separate entities
 - Data is modeled in terms of only attributes and relationships (but no operations)
 - Processes are modeled to show the 1) input data, 2) the transformation that occurs on that data, and 3) the resulting output data
- Object-oriented analysis
 - Focuses on the definition of classes and the manner in which they collaborate with one another to fulfill customer requirements

Elements of the Analysis Model

Object-oriented Analysis

Scenario-based modeling

Use case text
Use case diagrams
Activity diagrams
Swim lane diagrams

Structured Analysis

Flow-oriented modeling

Data structure diagrams
Data flow diagrams
Control-flow diagrams
Processing narratives

Class-based modeling

Class diagrams
Analysis packages
CRC models
Collaboration diagrams

Behavioral modeling

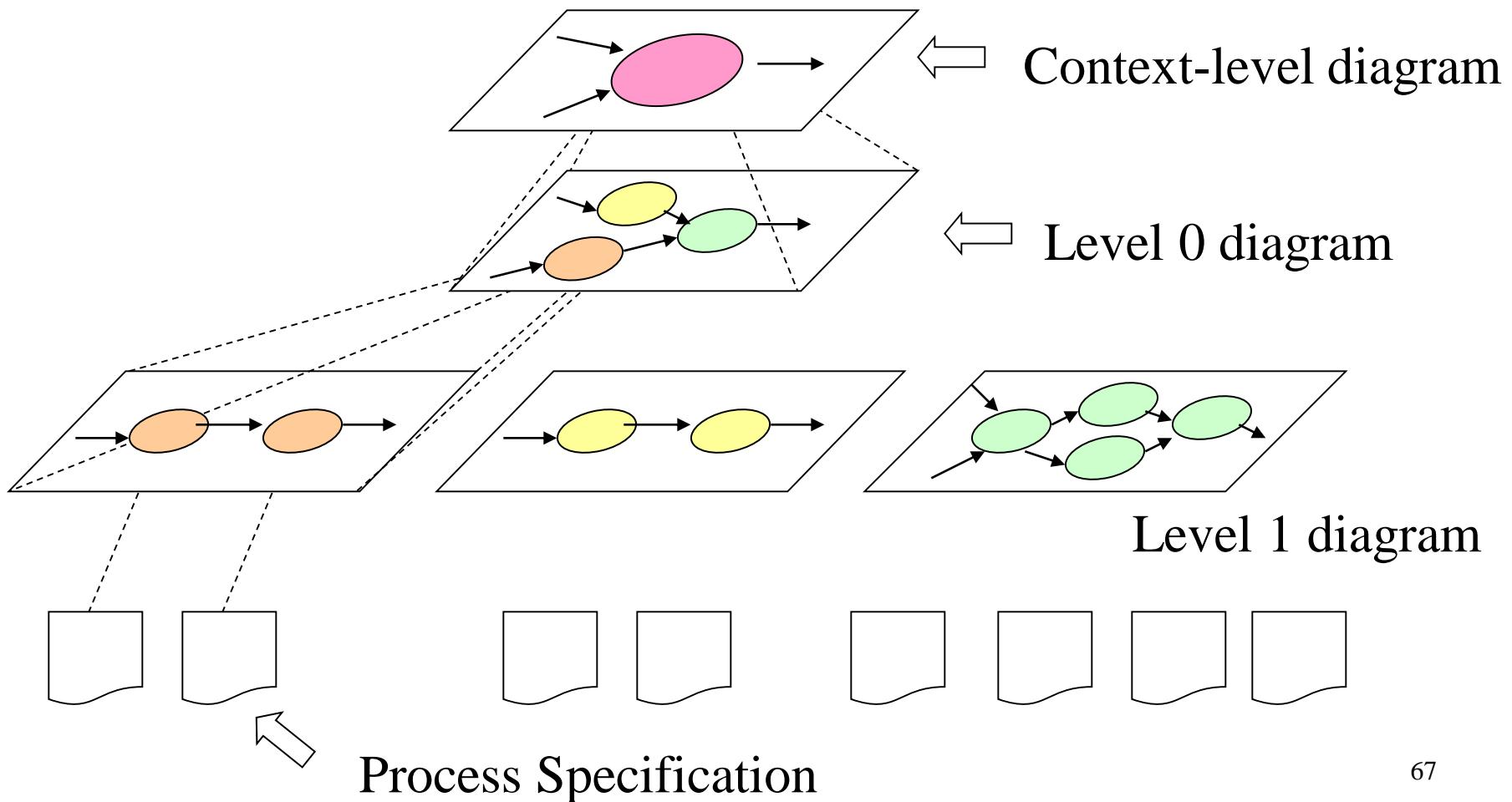
State diagrams
Sequence diagrams

Flow-oriented Modeling

Data Flow Diagrams

- A structured analysis **technique** that employs a set of visual **representations** of the **data** that moves through the organization, the **paths** through which the data moves, and the **processes** that produce, use, and transform data.

Diagram Layering and Process Refinement



Why Data Flow Diagrams?

- Can diagram the **organization** or the **system**
- Can diagram the **current** or **proposed** situation
- Can facilitate **analysis** or **design**
- Provides a good bridge from analysis to design
- Facilitates communication with the user at all stages

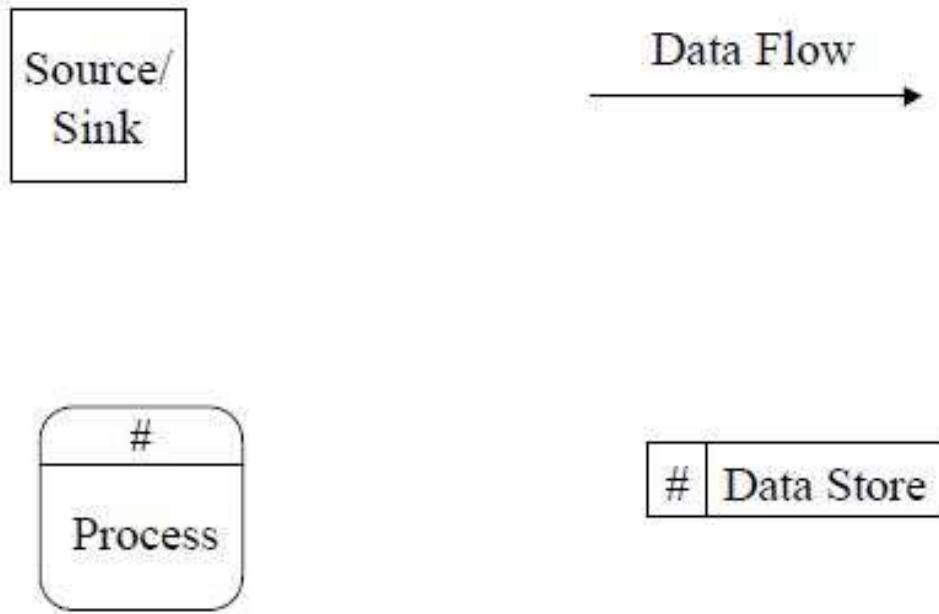
Types of DFDs

- **Current** - how data flows now
- **Proposed** - how we'd like it to flow
- **Logical** - the “essence” of a process
- **Physical** - the implementation of a process
- **Partitioned physical** - system architecture or high-level design

Levels of Detail

- **Context level diagram** - shows just the inputs and outputs of the system
- **Level 0 diagram** - decomposes the process into the major subprocesses and identifies what data flows between them
- **Child diagrams** - increasing levels of detail
- **Primitive diagrams** - lowest level of decomposition

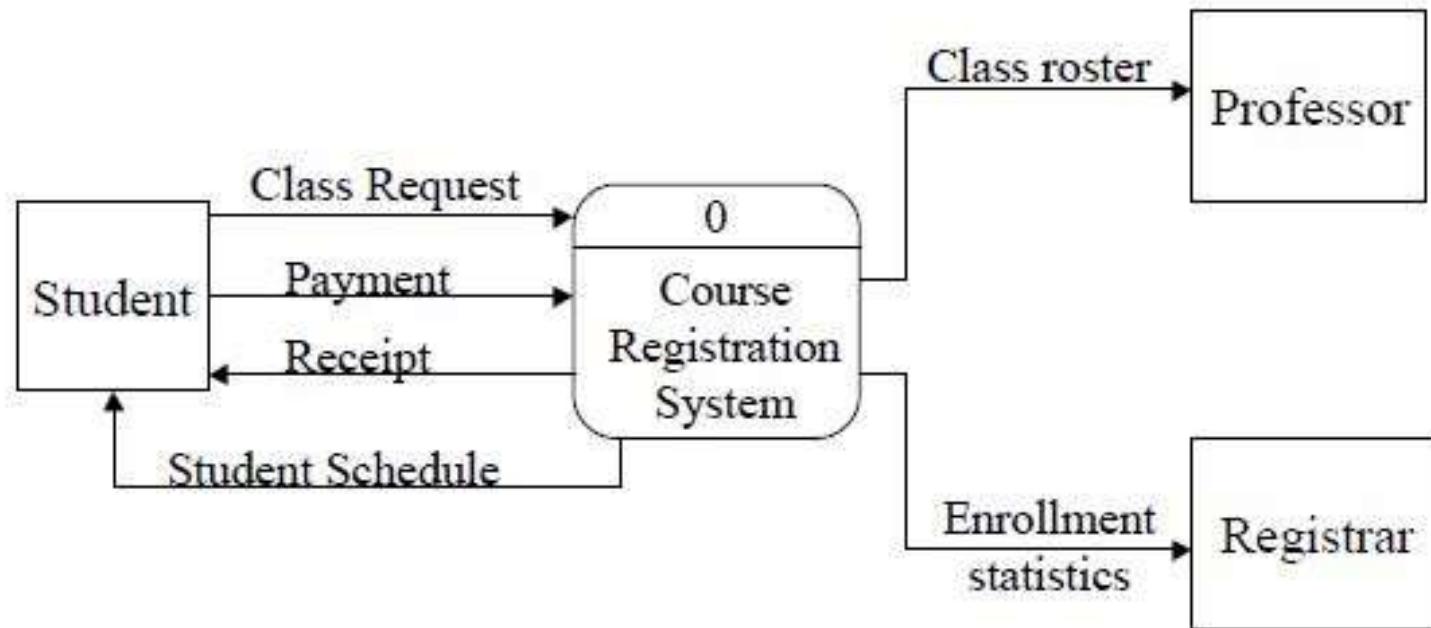
Four Basic Symbols



Context Level Diagram

- Just one process
- All sources and sinks that provide data to or receive data from the process
- Major data flows between the process and all sources/sinks
- No data stores

Course Registration: Context level Diagram

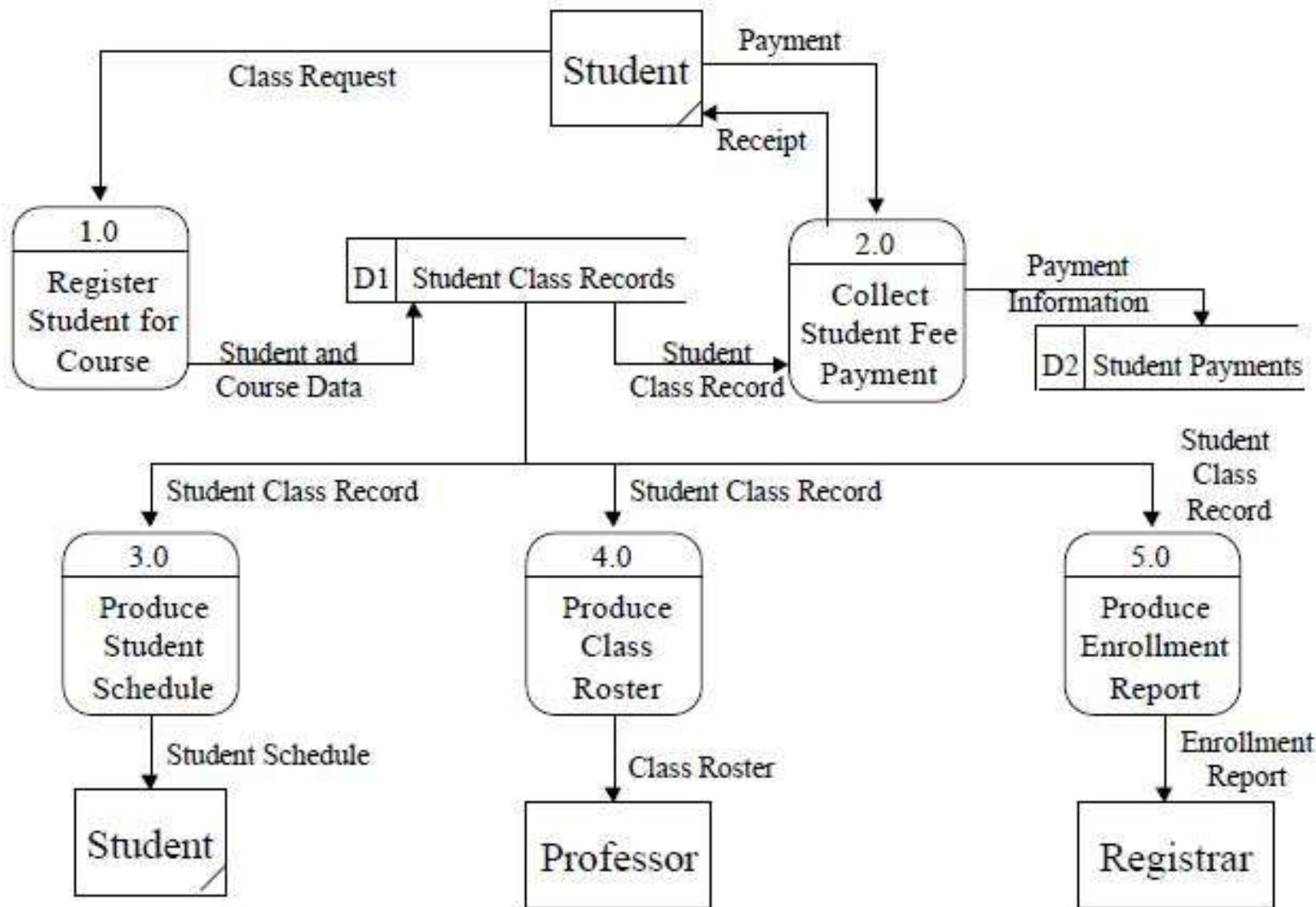


<https://userpages.umbc.edu/~cseaman/ifsm636/lect0913.pdf>

Level 0 Diagram

- Process is “exploded”
- Sources, sinks, and data flows repeated from context diagram
- Process broken down into subprocesses, numbered sequentially
- Lower-level data flows and data stores added

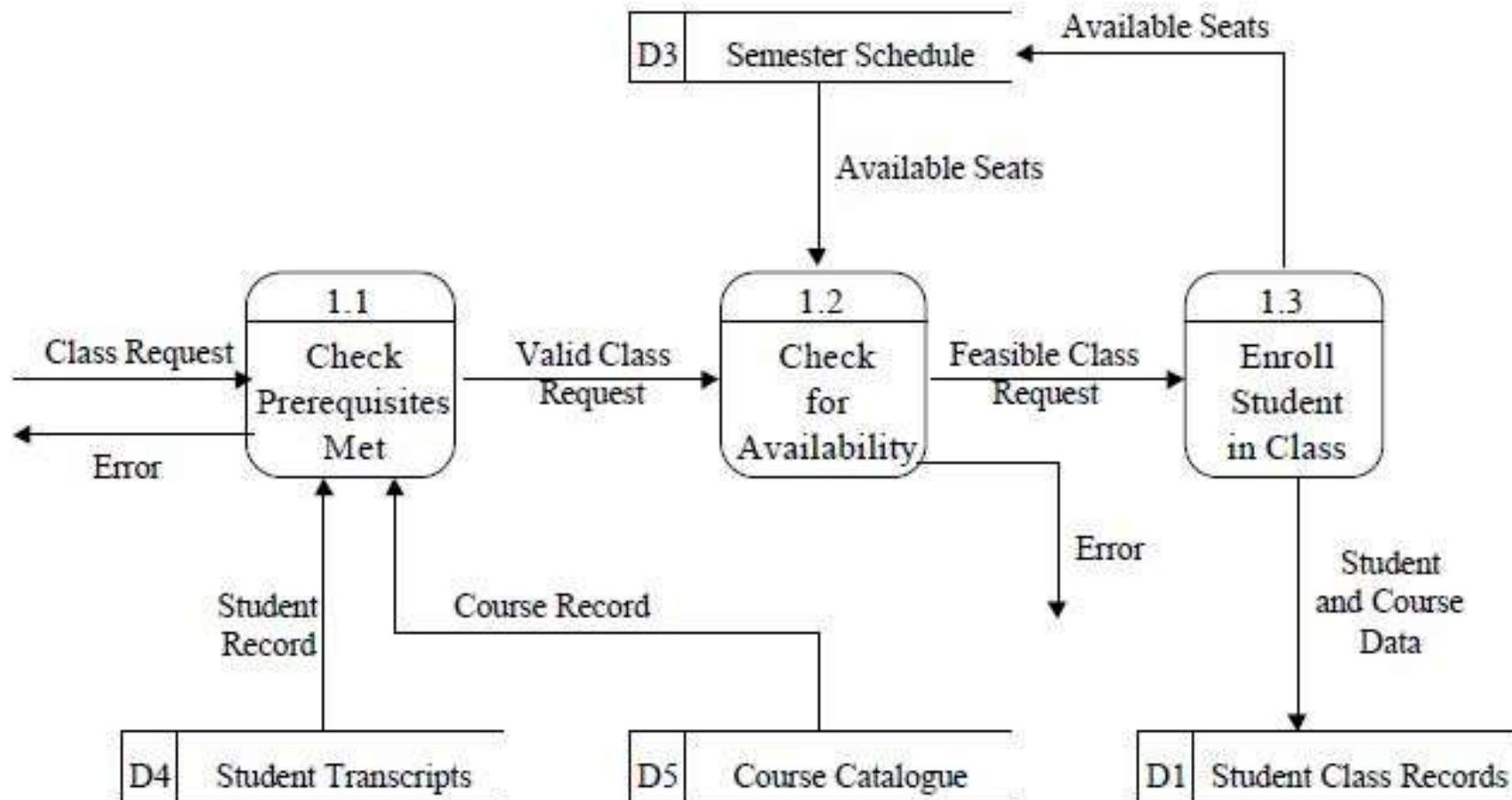
Course Registration: Current Logical Level 0 Diagram



Child Diagrams

- “Explode” one process in level 0 diagram
- Break down into lower-level processes, using numbering scheme
- Must include all data flow into and out of “parent” process in level 0 diagram
- **Don’t include sources and sinks**
- May add lower-level data flows and data stores

Course Registration: Current Logical Child Diagram (or Level 1) - Student registration in a course



Physical DFDs

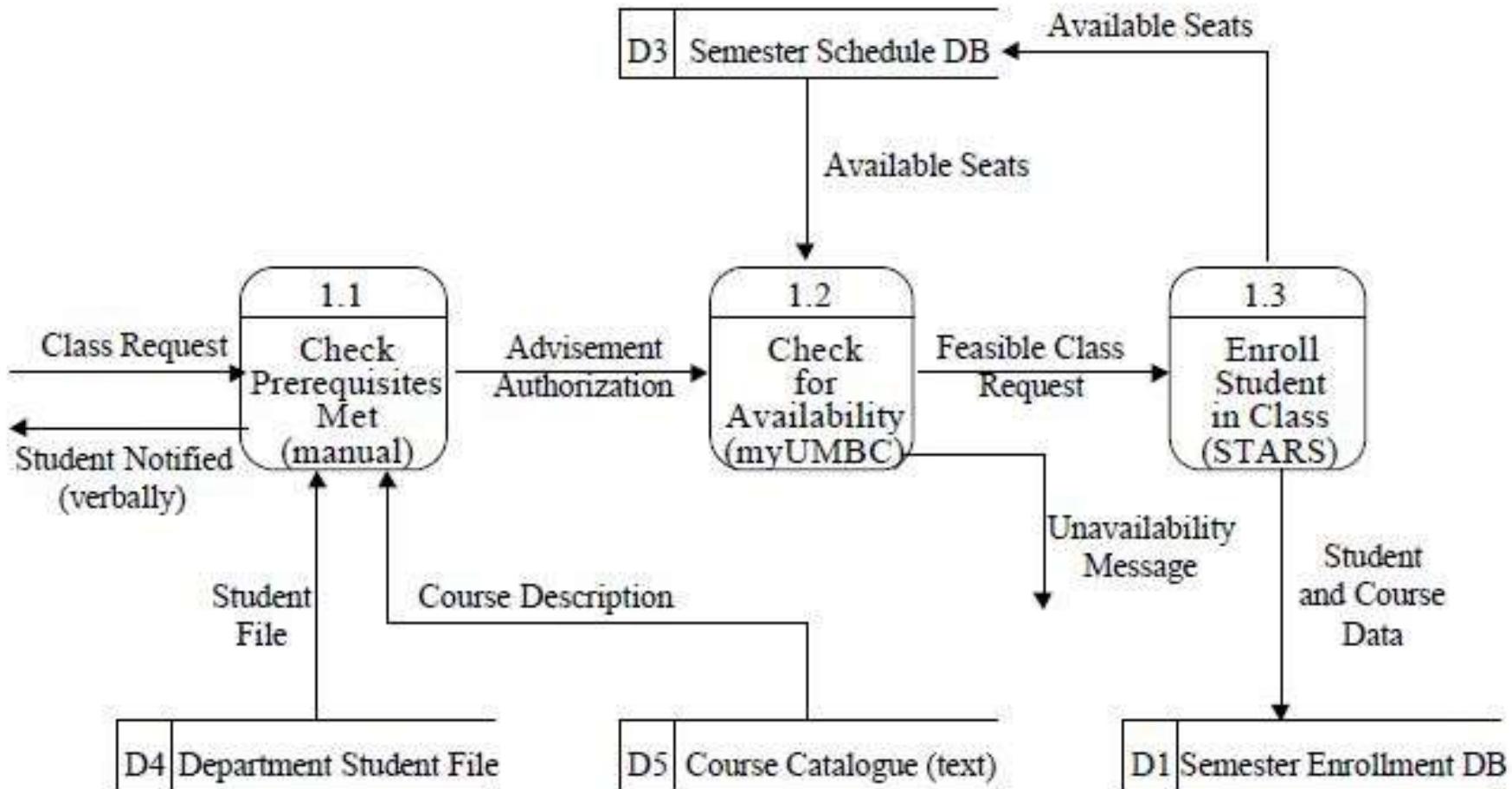
- Model the **implementation** of the system
- Start with a set of child diagrams or with level 0 diagram
- Add implementation details
 - indicate manual vs. automated processes
 - describe form of data stores and data flows
 - extra processes for maintaining data

A logical DFD focuses on the business and business activities, while a physical DFD looks at how a system is implemented.

the logical diagram provides the “what” and the physical provides the “how”

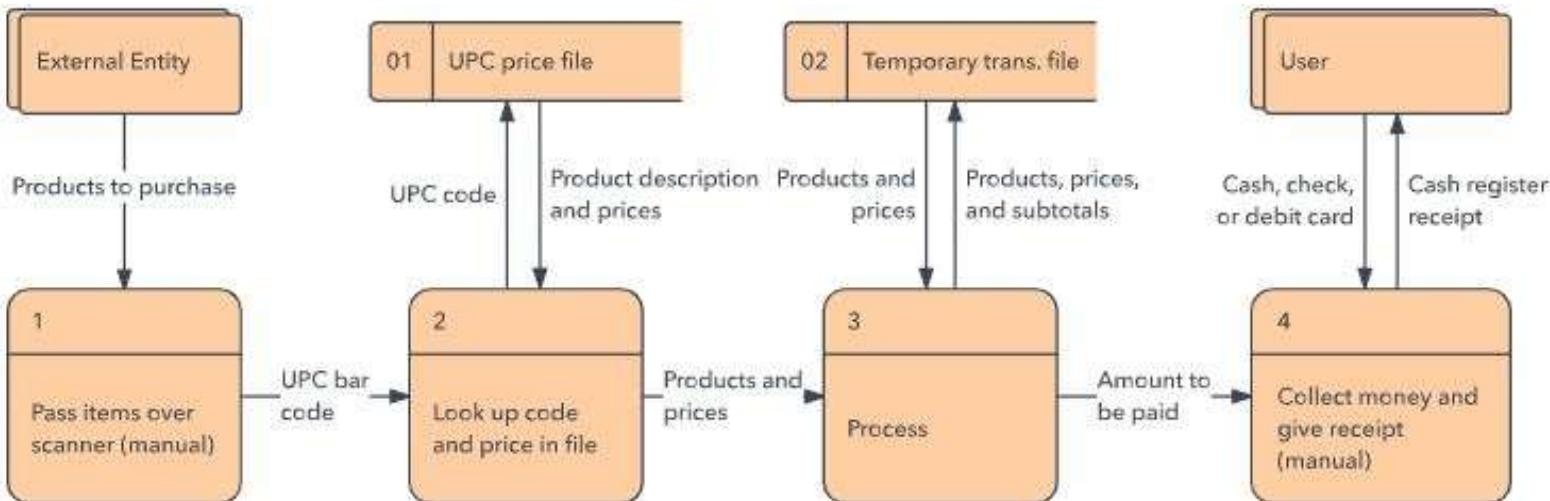
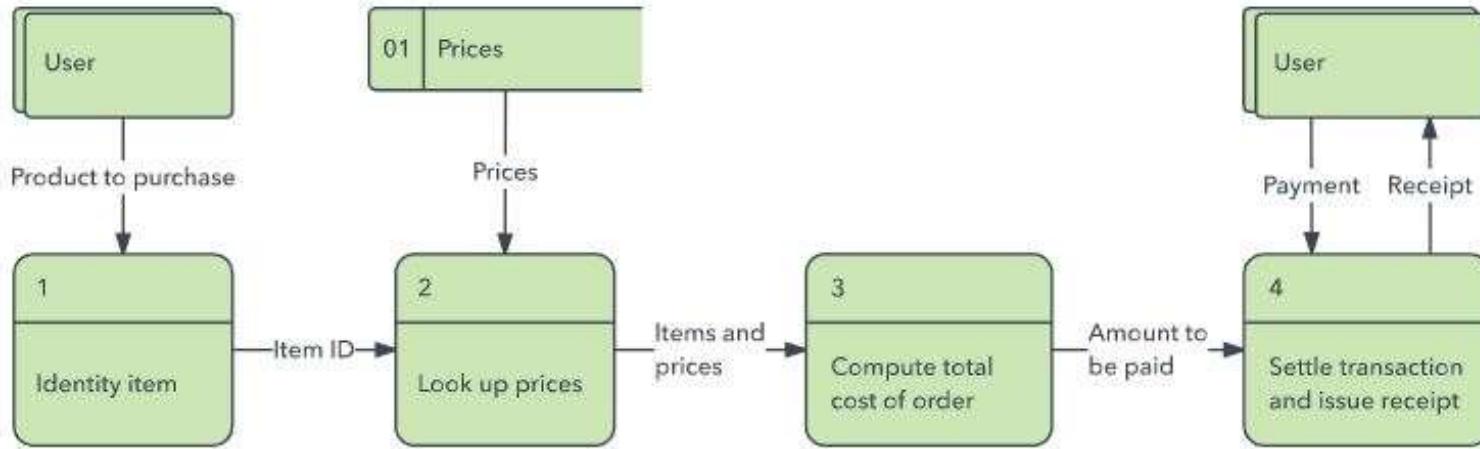
The logical DFD describes the business events that take place and the data required for each event. It provides a solid basis for the physical DFD, which depicts how the data system will work, such as the hardware, software, paper files and people involved.

Course Registration: Current Physical Child Diagram



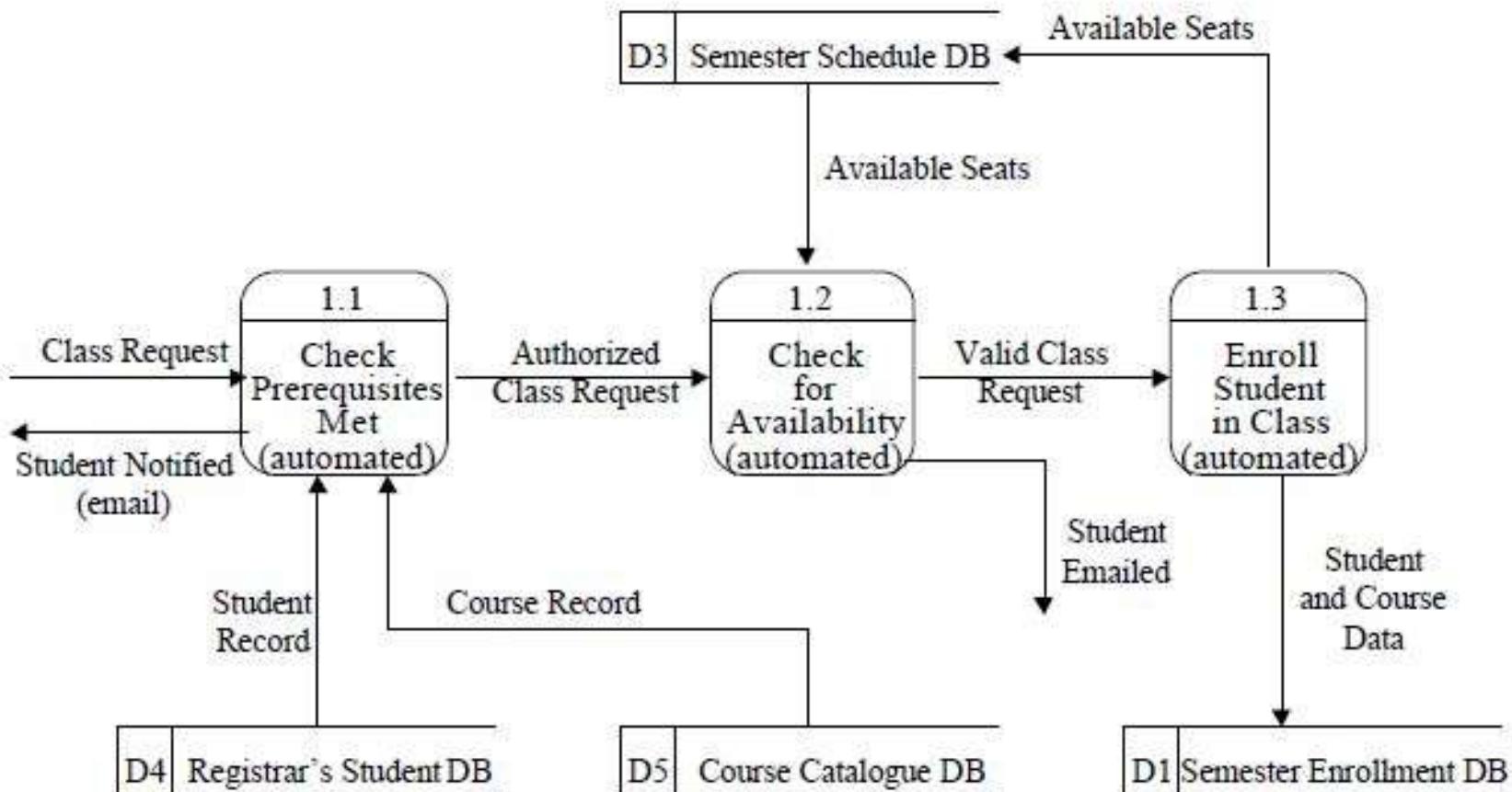
Logical vs Physical DFD

Logical DFD



Physical DFD

Course Registration: Proposed Physical Child Diagram



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

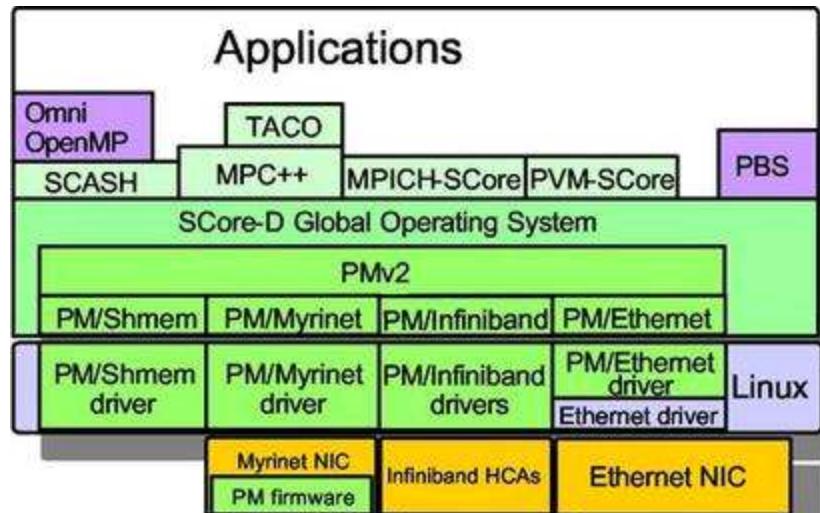
- A software architecture enables a software engineer to
 - Analyze the effectiveness of the design in meeting its stated requirements
 - Consider architectural alternatives at a stage when making design changes is still relatively easy
 - Reduce the risks associated with the construction of the software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

Importance of Software Architecture

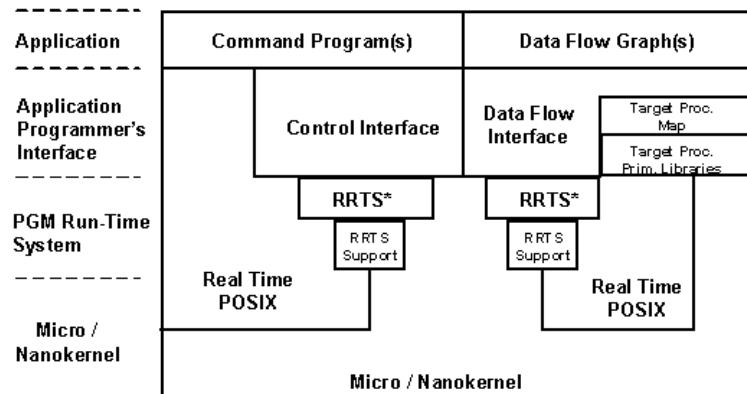
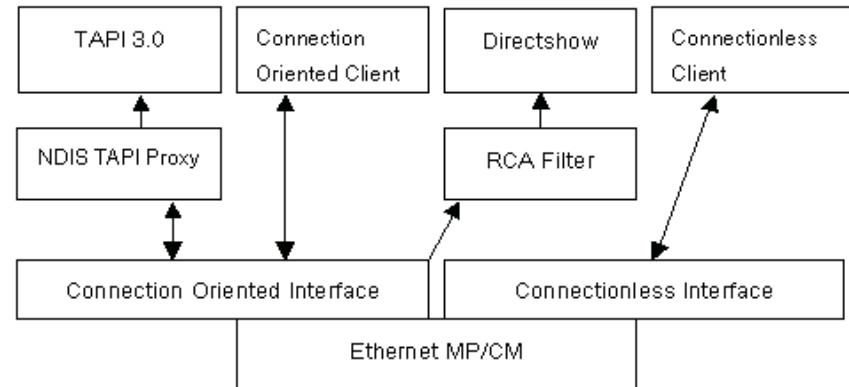
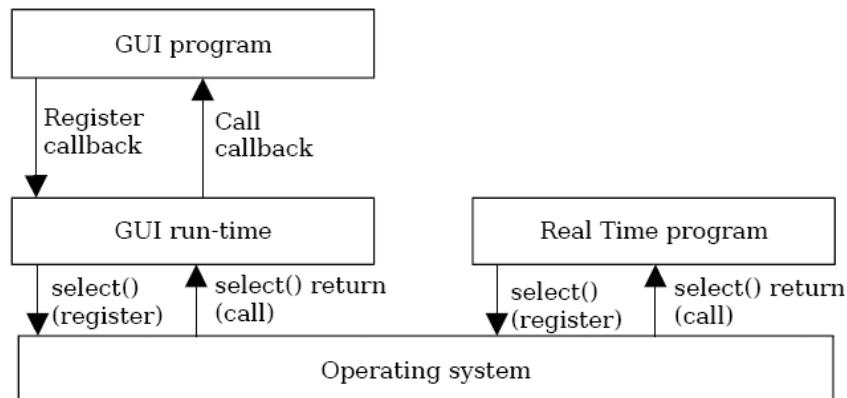
[Bass]

- 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



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

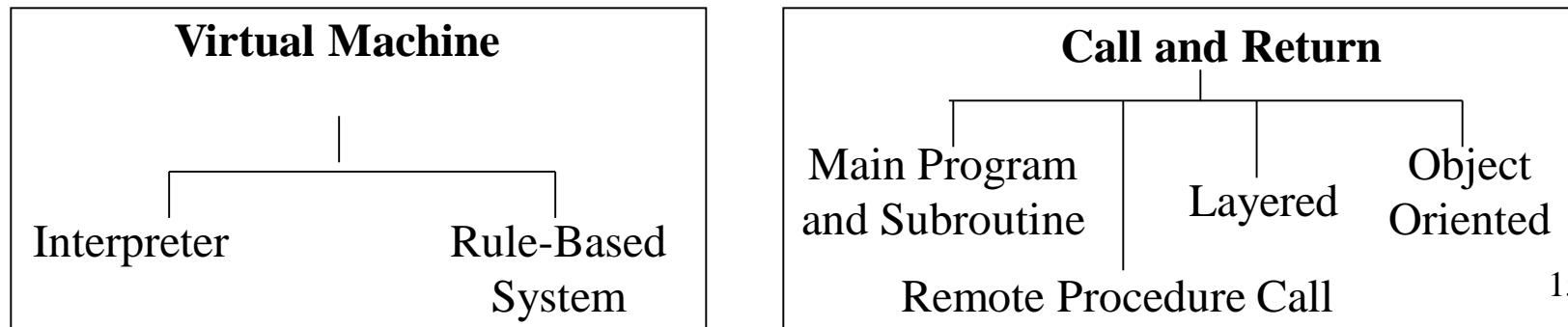
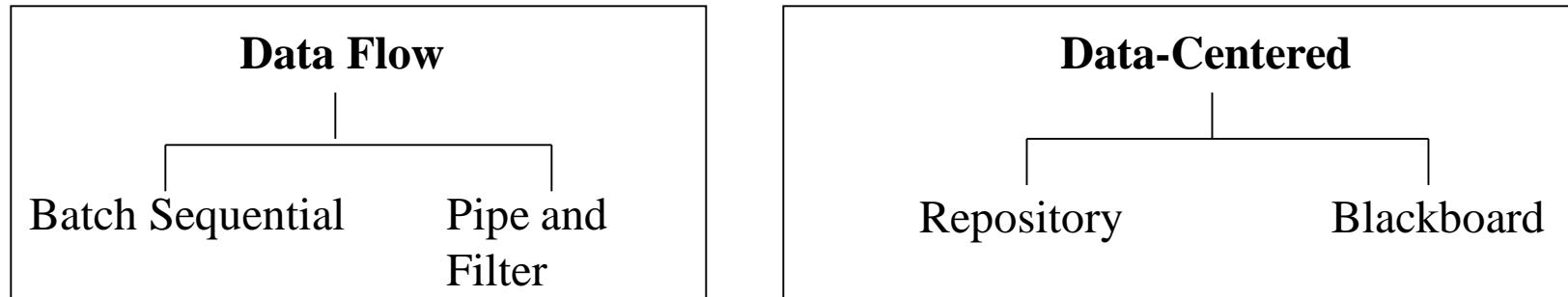
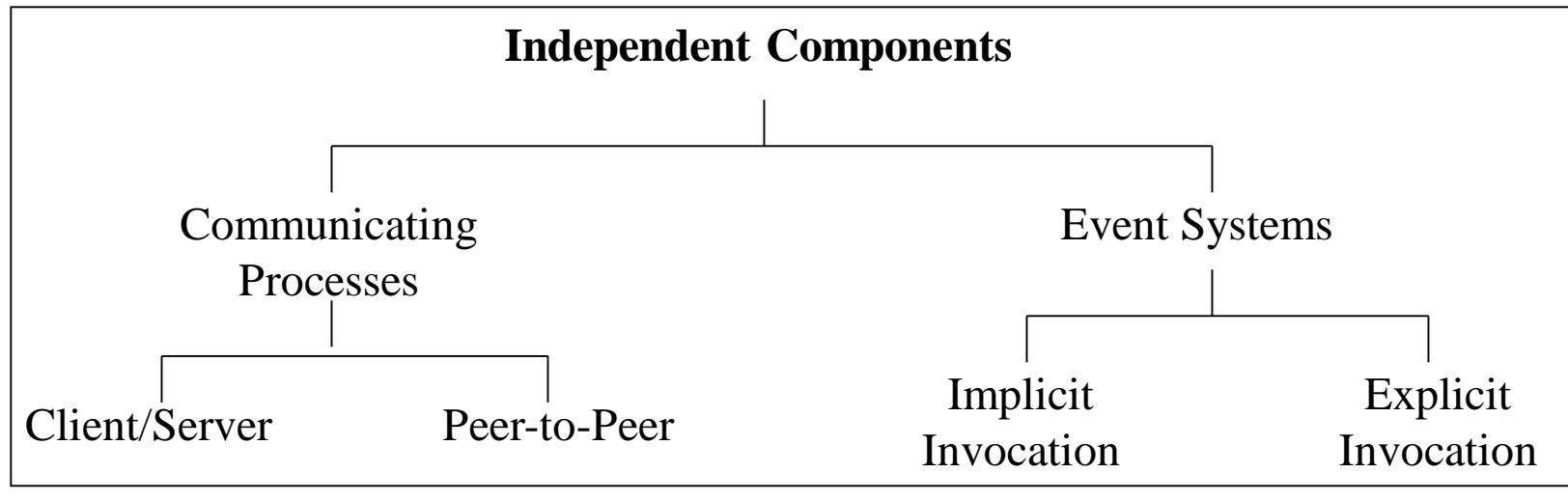
Pueblo

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



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 simply exist to move data between filters
(More on next slide)

Data Flow Style (continued)

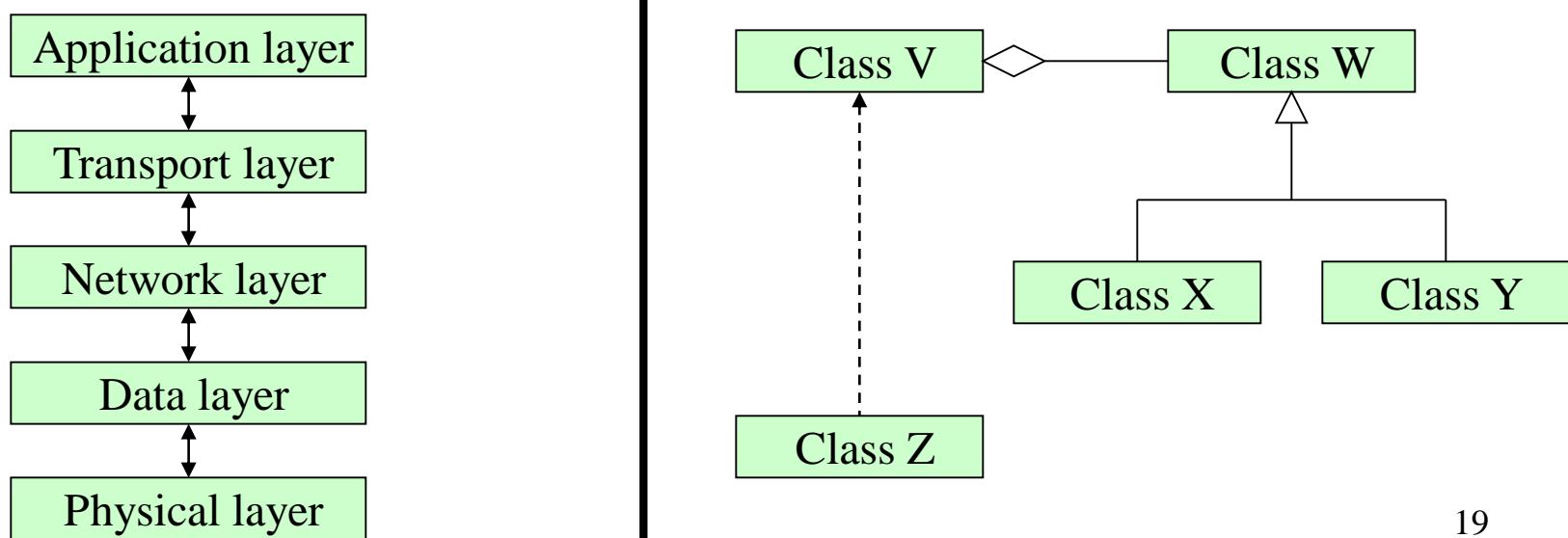
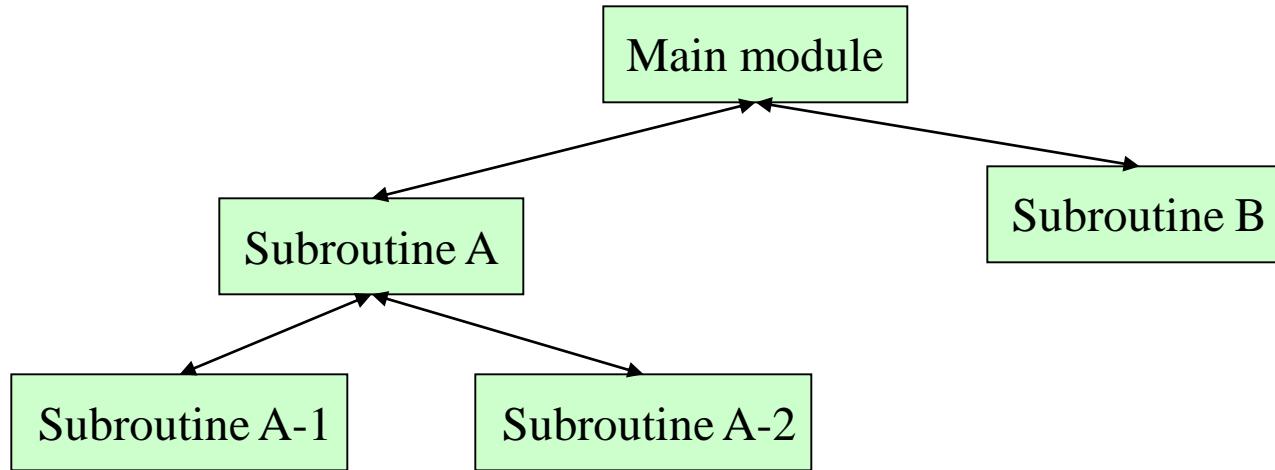
- Advantages
 - Has a simplistic design in the limited ways in which the components interact with the environment
 - Consists of no more and no less than the construction of its parts
 - Simplifies reuse and maintenance
 - Is easily made into a parallel or distributed execution in order to enhance system 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



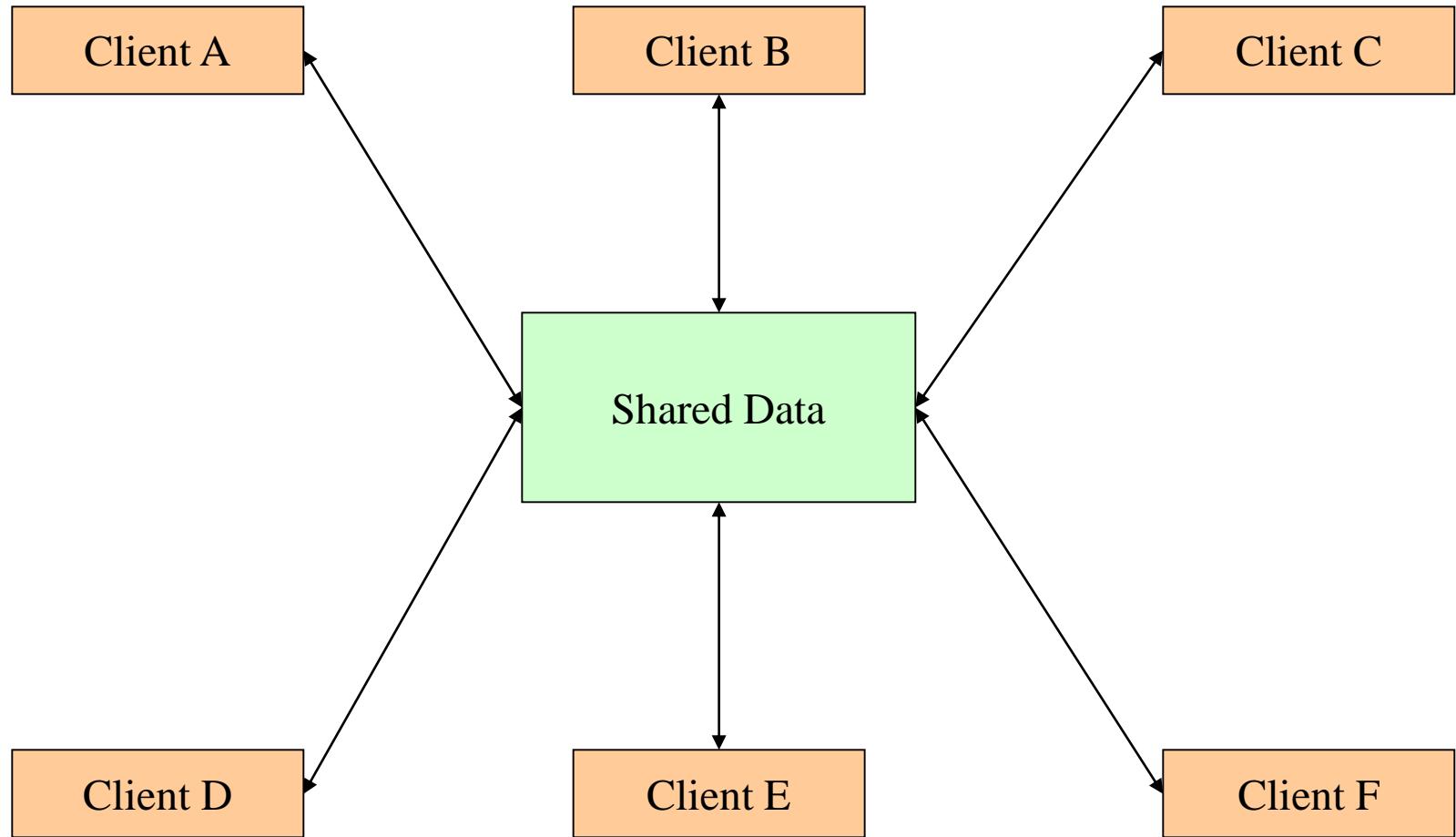
Call-and-Return Style

- Has the goal of modifiability and scalability
- Has been the dominant architecture since the start of software development
- Main program and subroutine style
 - Decomposes a program hierarchically into small pieces (i.e., modules)
 - Typically has a single thread of control that travels through various components in the hierarchy
- Remote procedure call style
 - Consists of main program and subroutine style of system that is decomposed into parts that are resident on computers connected via a network
 - Strives to increase performance by distributing the computations and taking advantage of multiple processors
 - Incurs a finite communication time between subroutine call and response
 - Remote File/database access
 - (More on next slide)

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



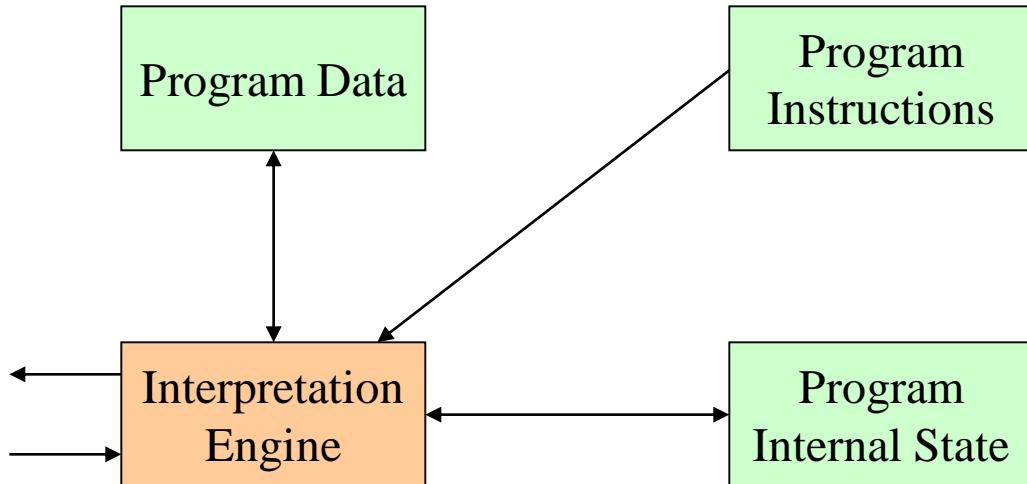
Data-Centered Style (continued)

- 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

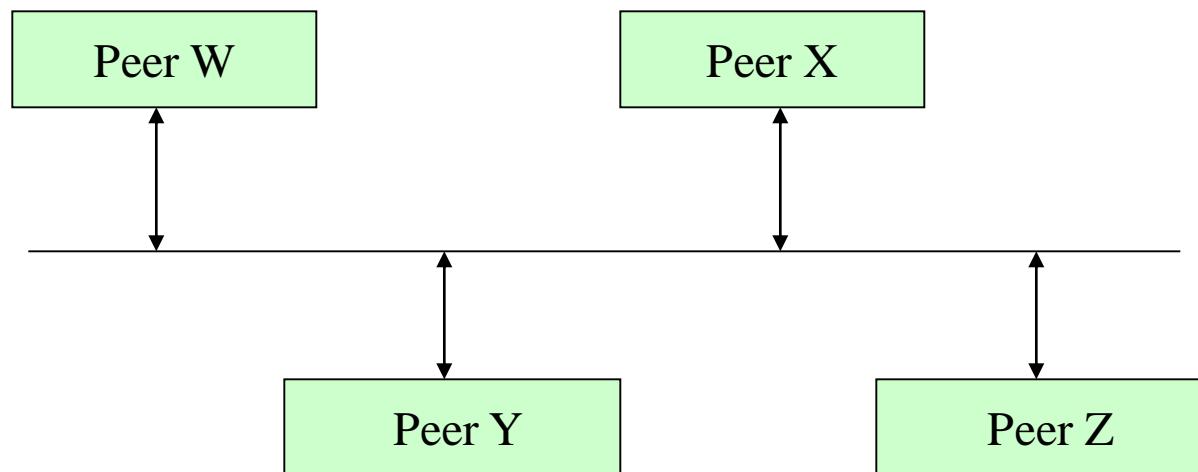
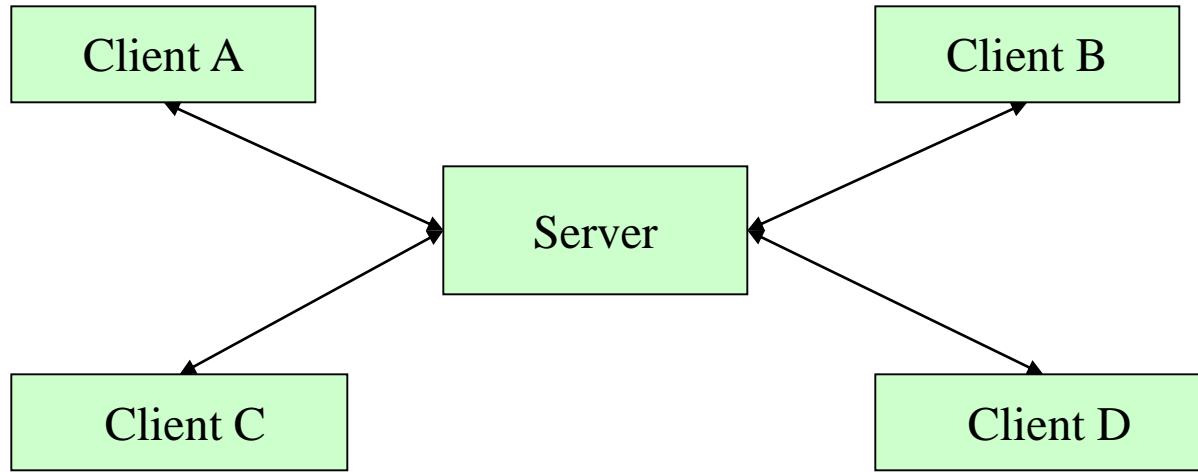
Virtual Machine Style



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 of computation but have no make of machine to directly run it on

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

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

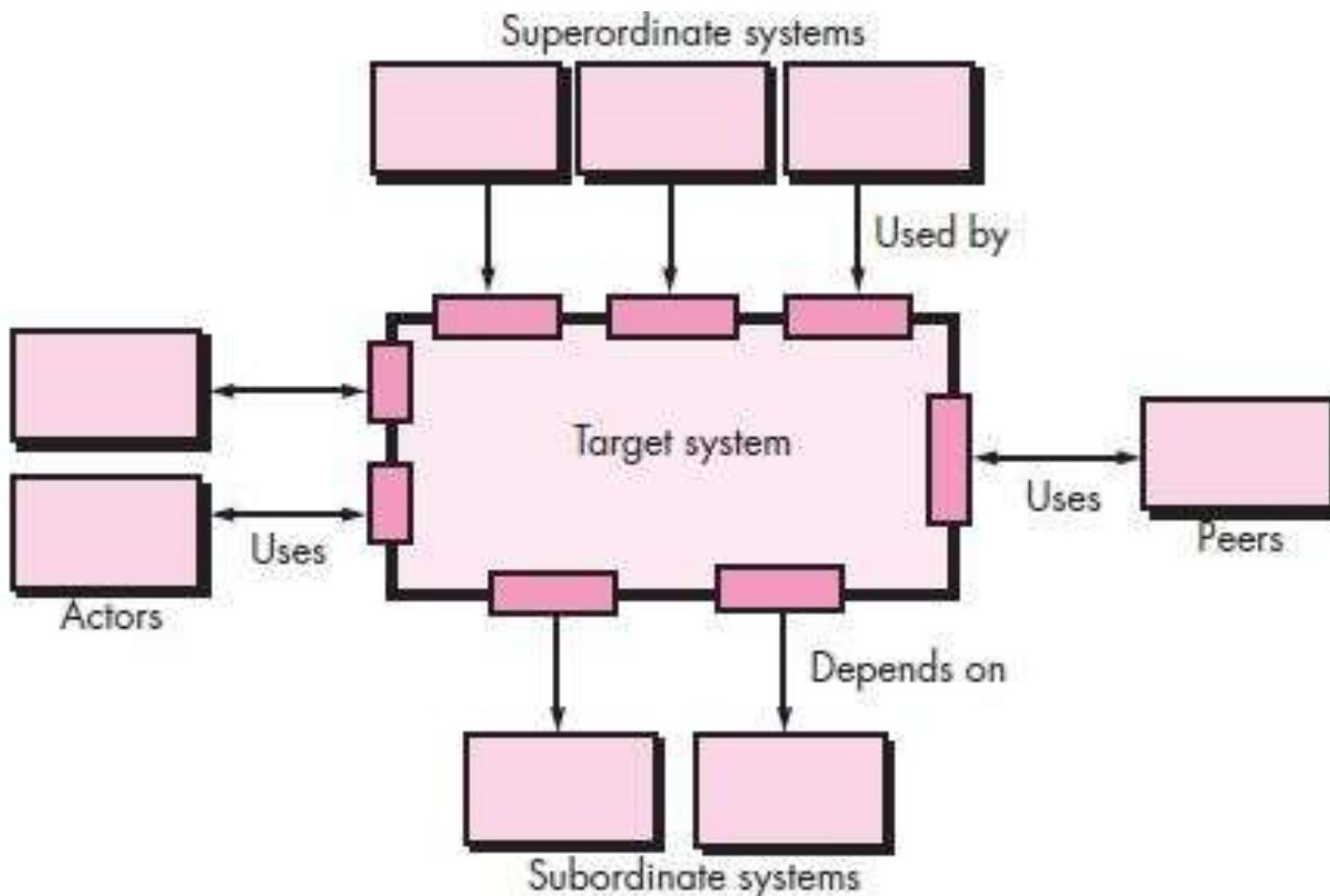
Architectural Design Process

Architectural Design Steps

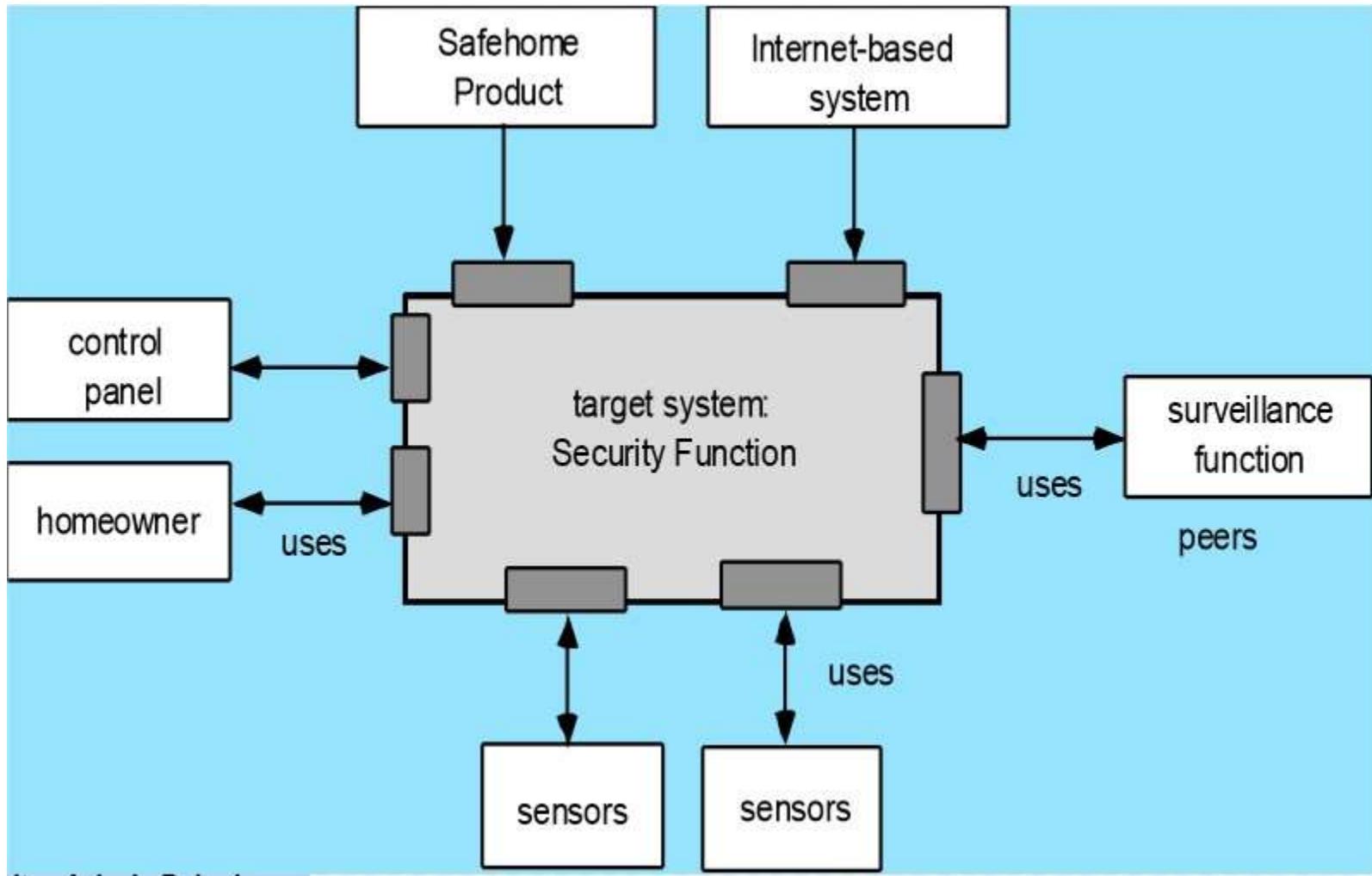
- 1) Represent the system in context
- 2) Define archetypes
- 3) Refine the architecture into components
- 4) Describe instantiations of the system

"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

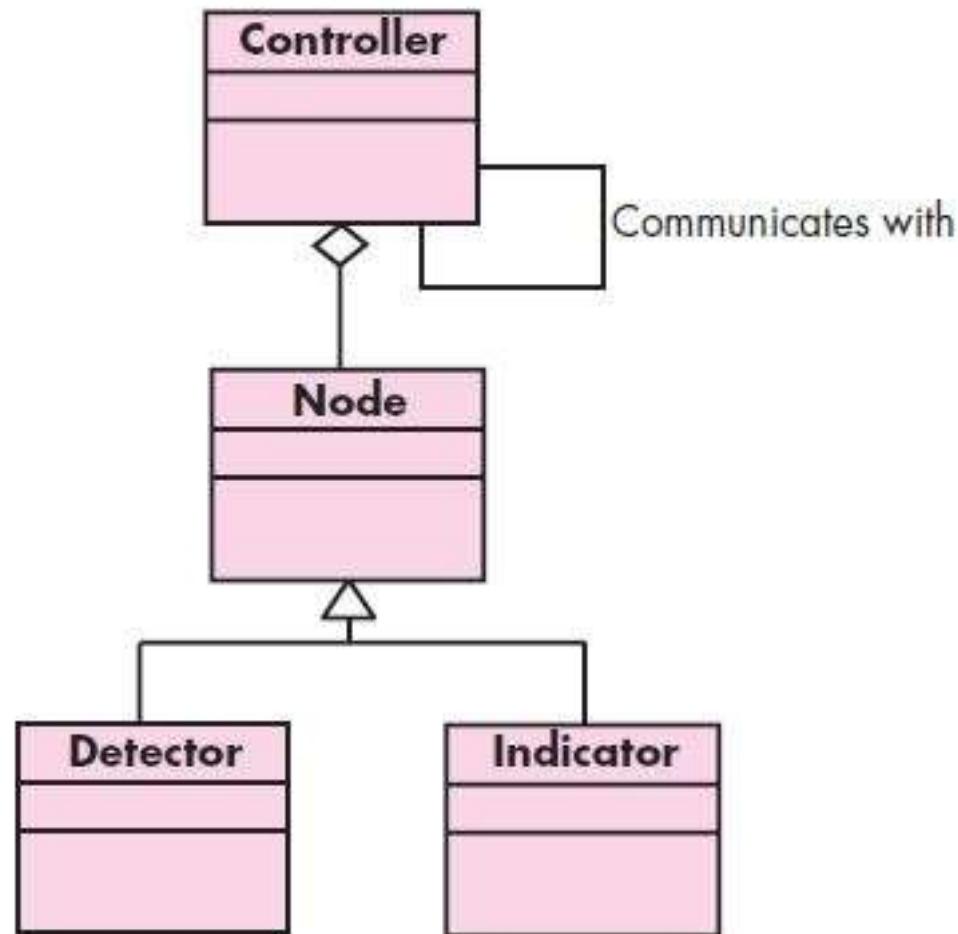
Example Archetypes in Software Architecture

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

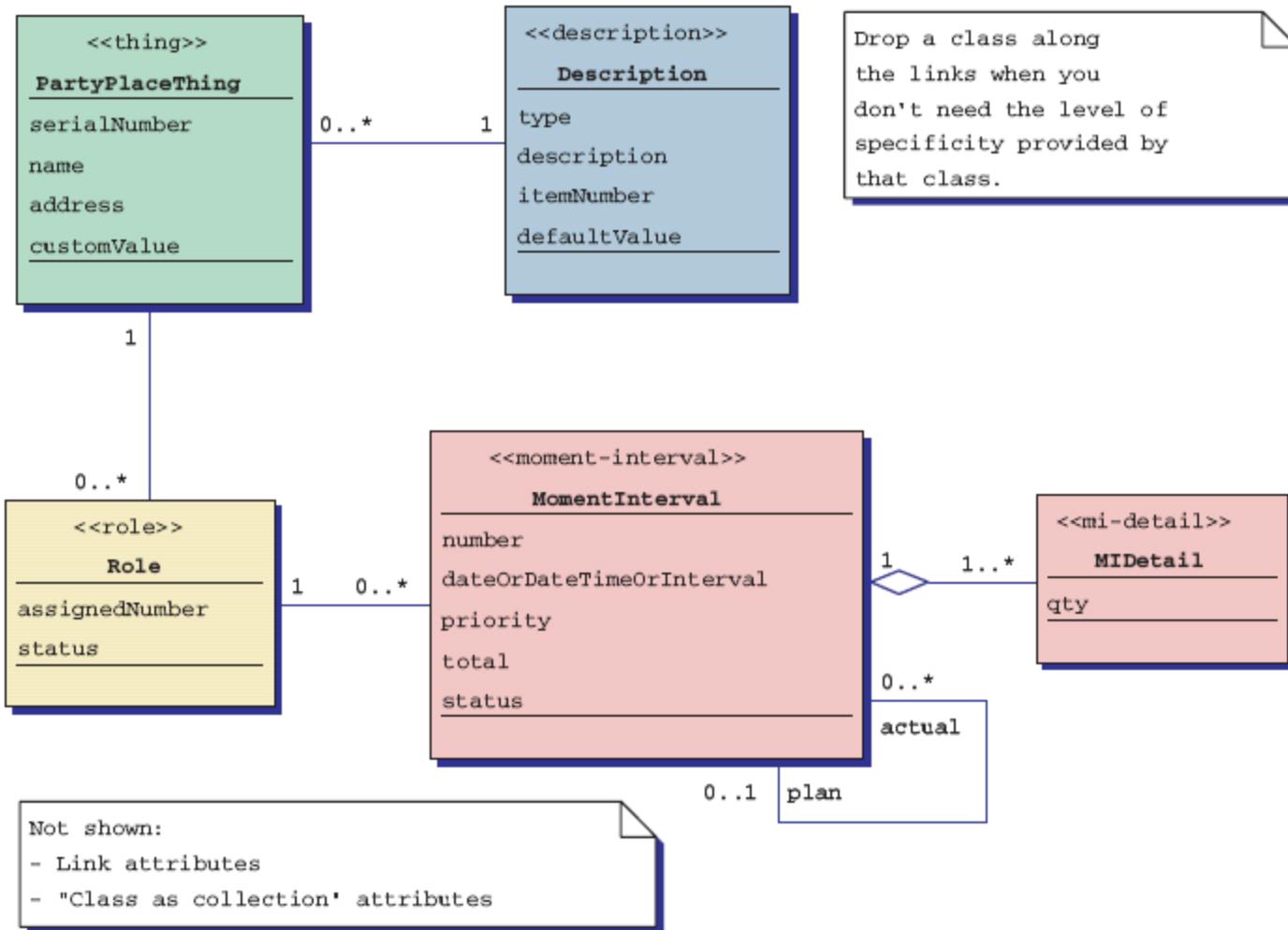
(Source: Pressman)

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

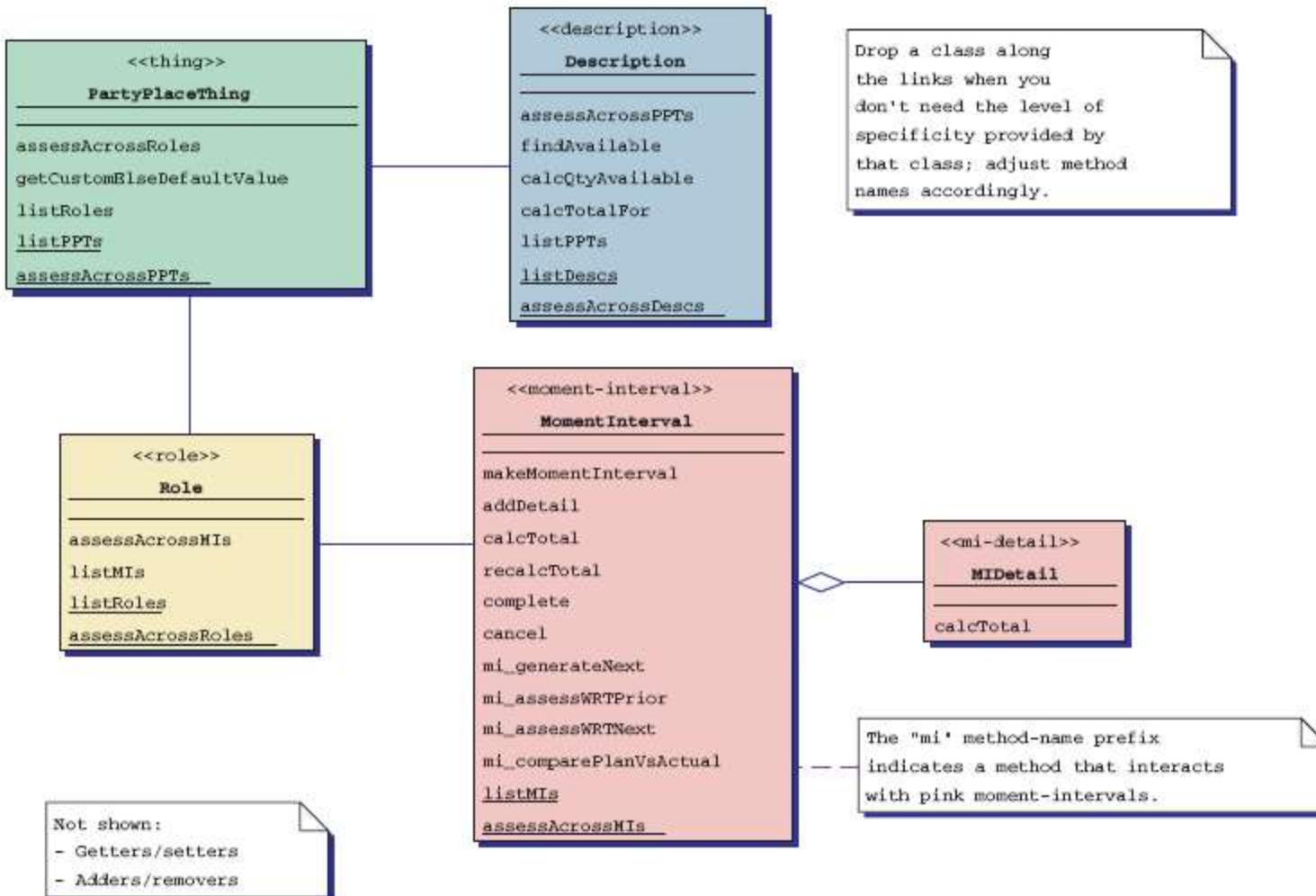
UML relationships for *SafeHome* security function archetypes



Archetypes – their attributes



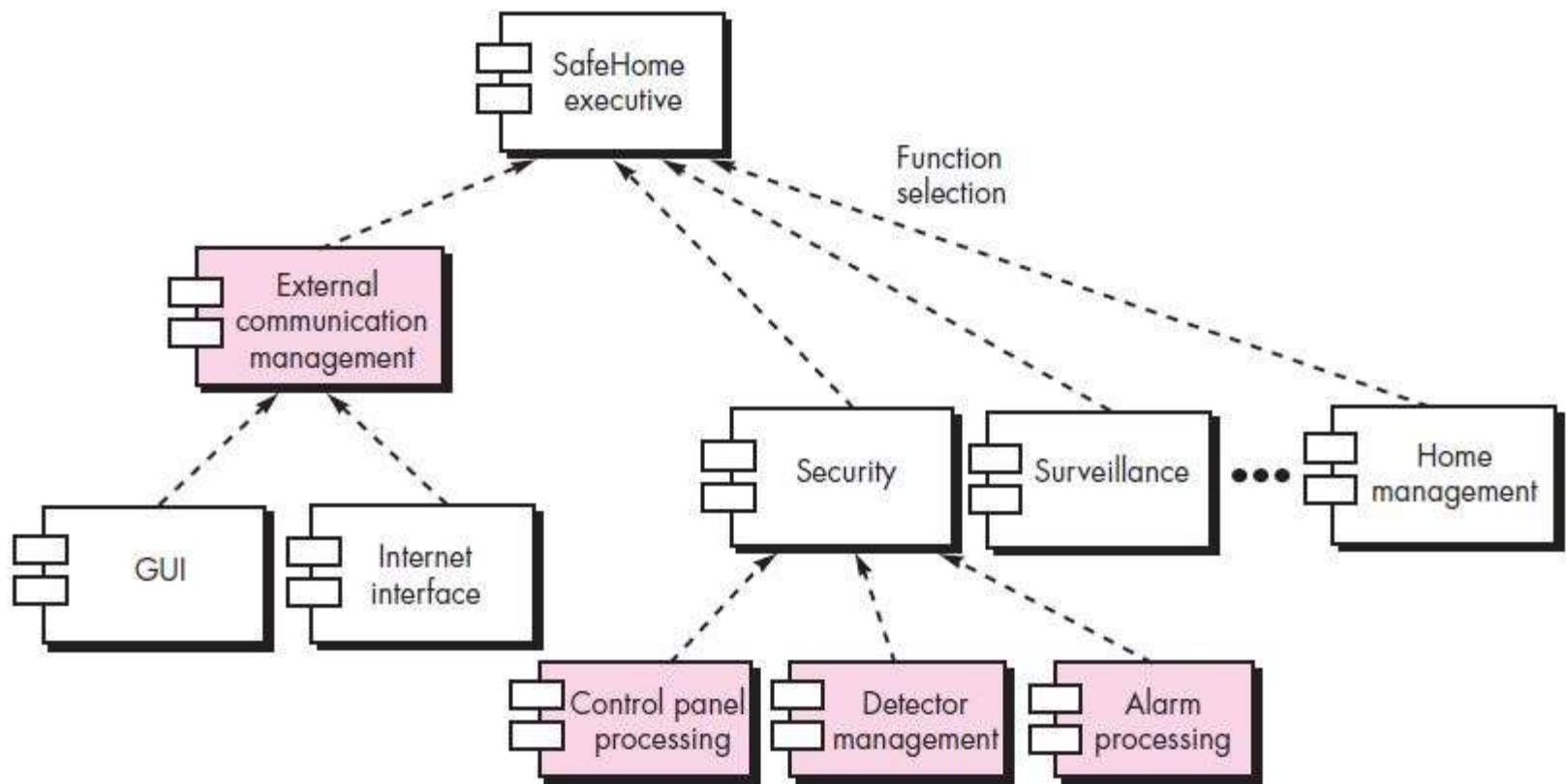
Archetypes – their methods



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

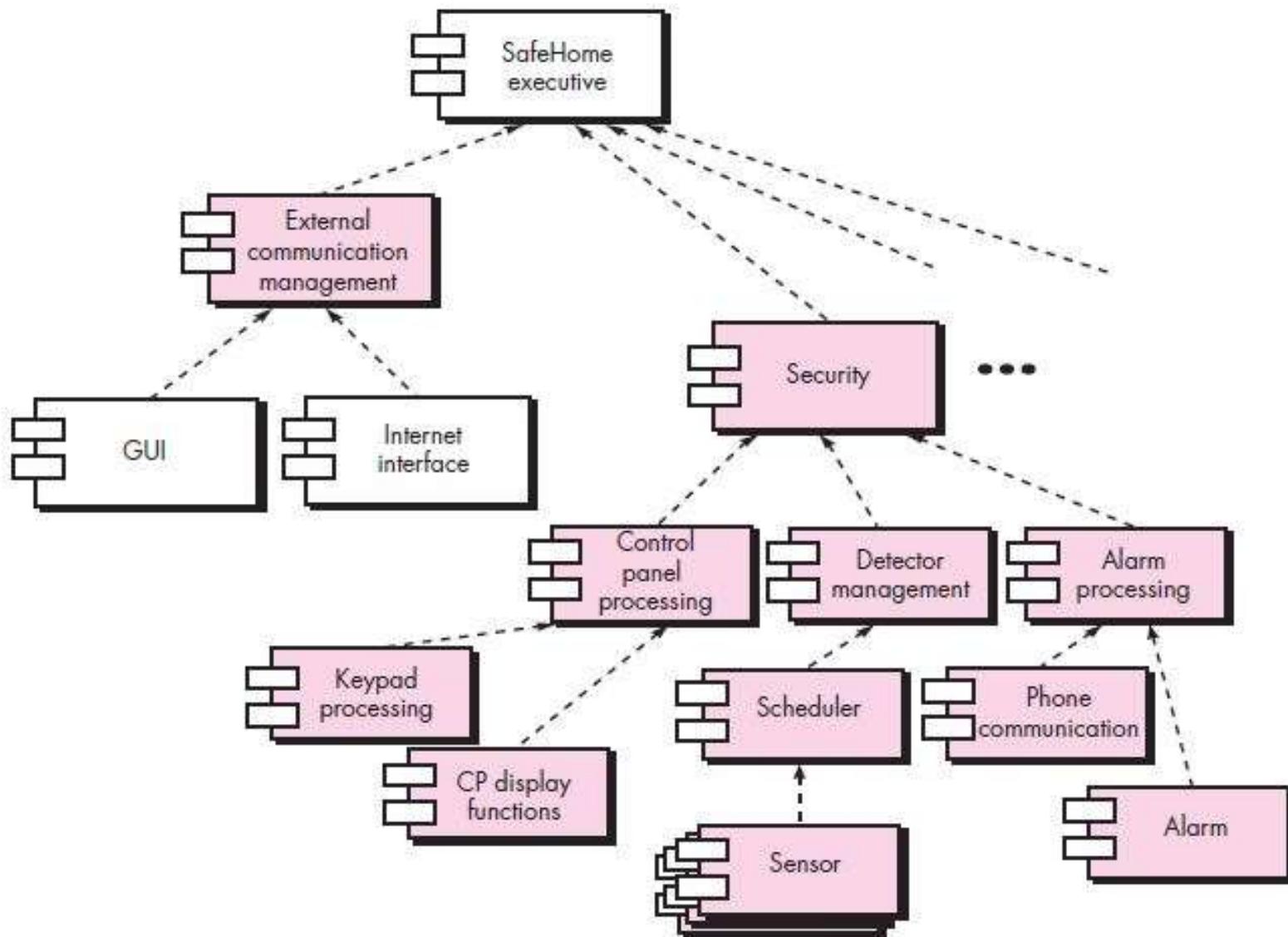
Overall architectural structure for *SafeHome* with top-level components



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 $U \leftarrow \rightarrow \square \leftarrow \rightarrow V$
 - Represents a dependency relationship among consumers who use the same source or producer
 - Flow dependency $\rightarrow U \rightarrow V \rightarrow$
 - 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.” 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

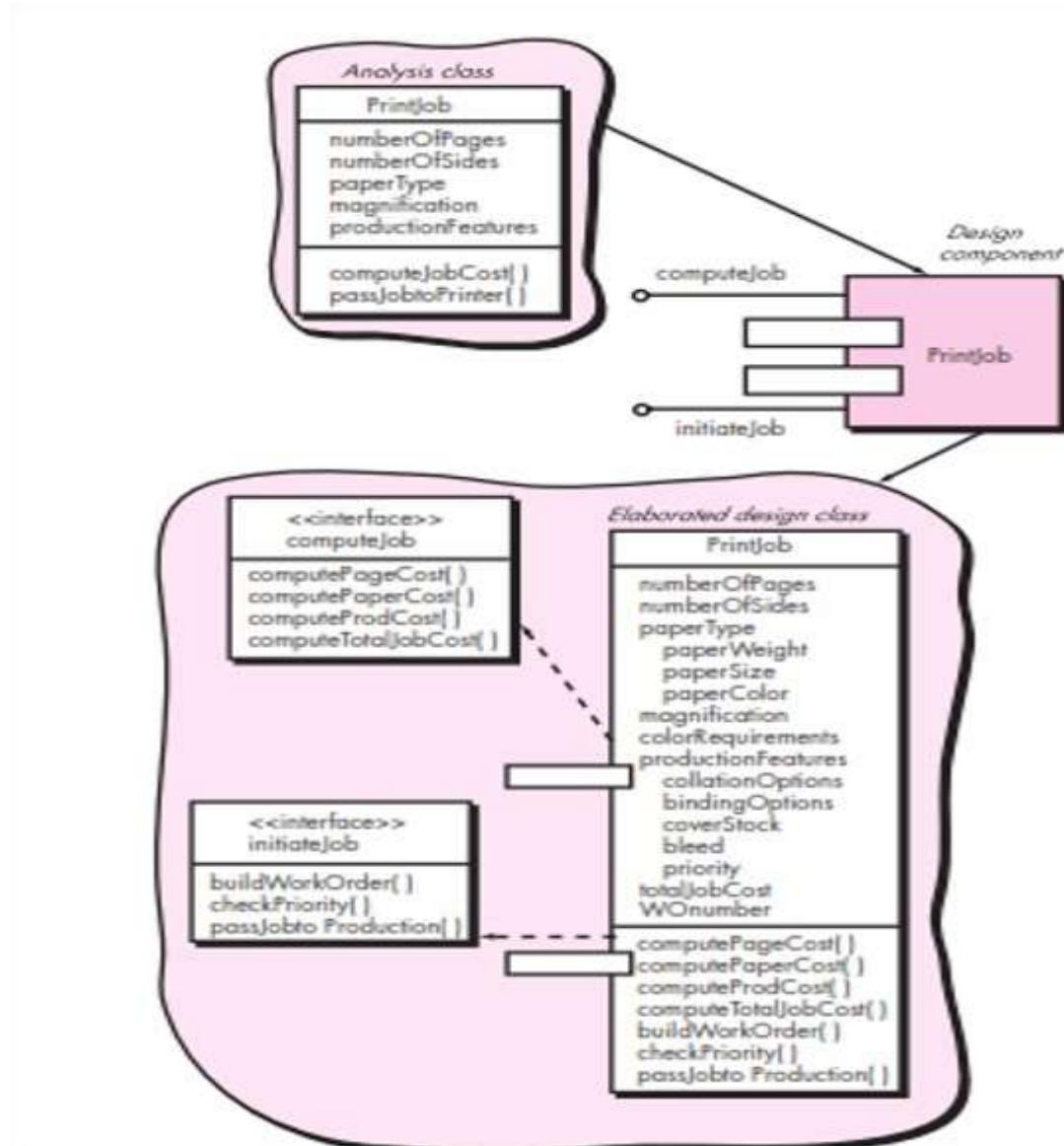
Defined

- 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
 - 1) Provide further elaboration of each attribute, operation, and interface
 - 2) Specify the data structure appropriate for each attribute
 - 3) Design the algorithmic detail required to implement the processing logic associated with each operation
 - 4) Design the mechanisms required to implement the interface to include the messaging that occurs between objects

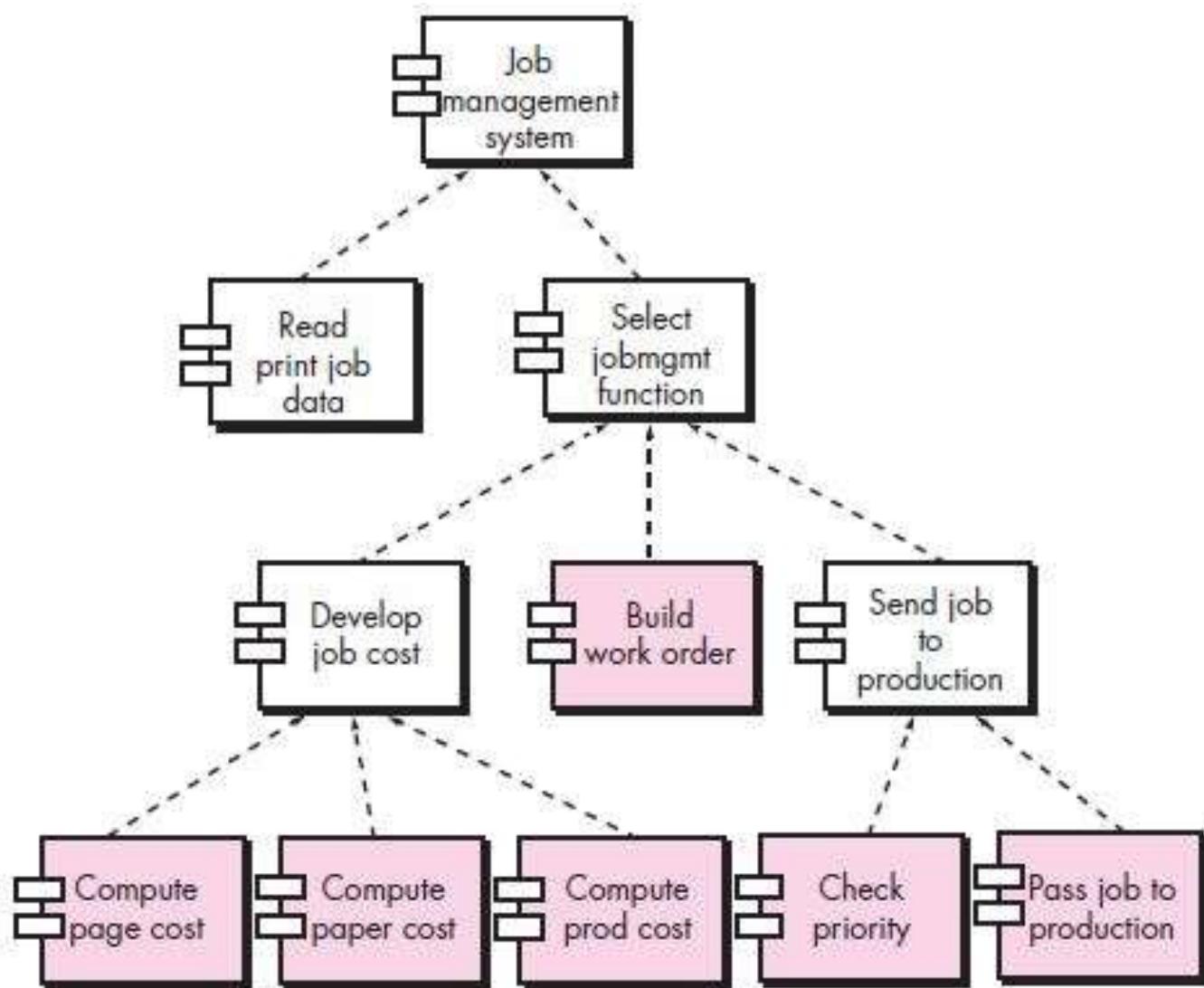
Elaboration of a design component



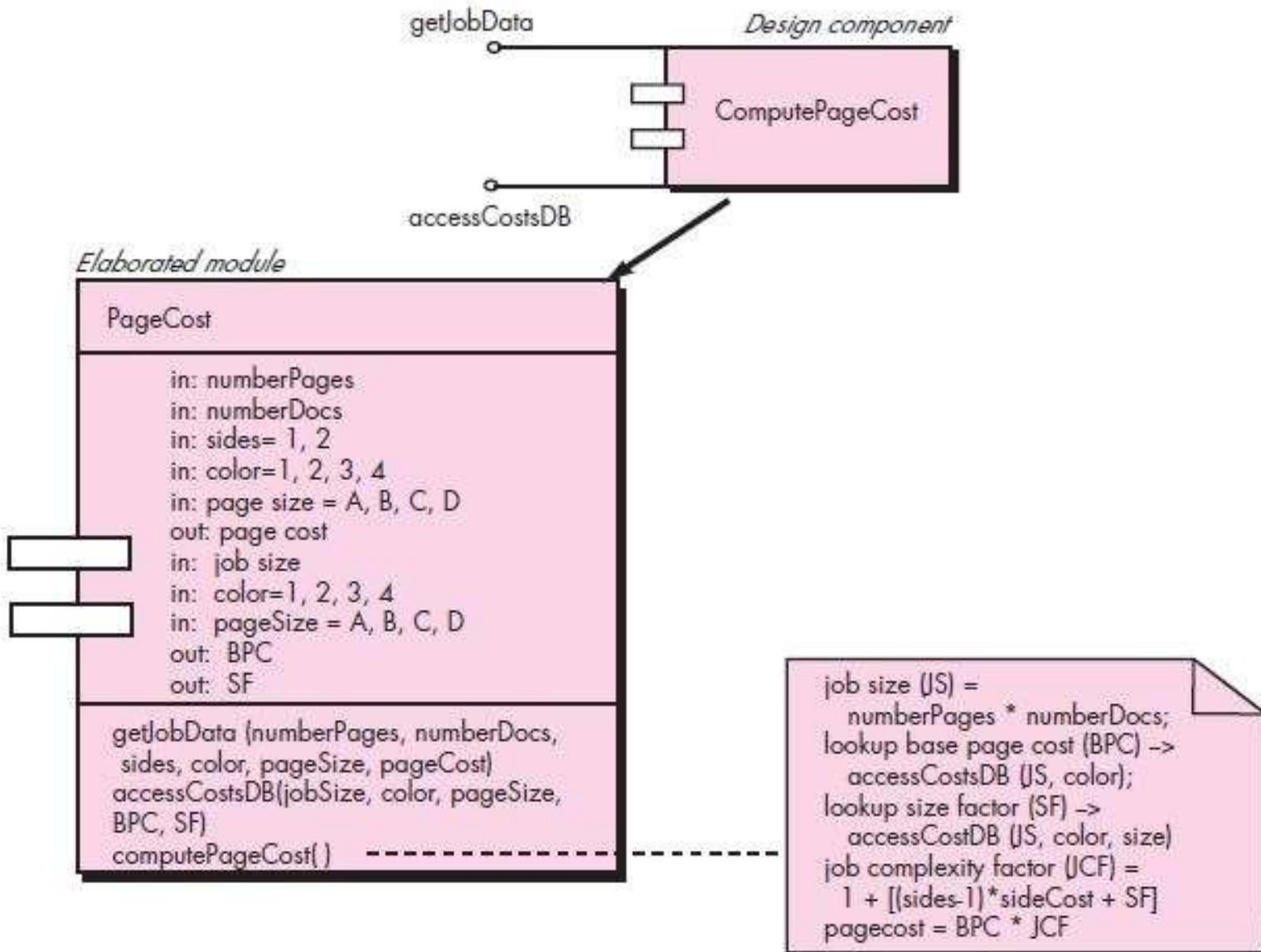
Conventional/Traditional View

- 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
 - 1) Define the interface for the transform (the order, number and types of the parameters)
 - 2) Define the data structures used internally by the transform
 - 3) Design the algorithm used by the transform (using a stepwise refinement approach)

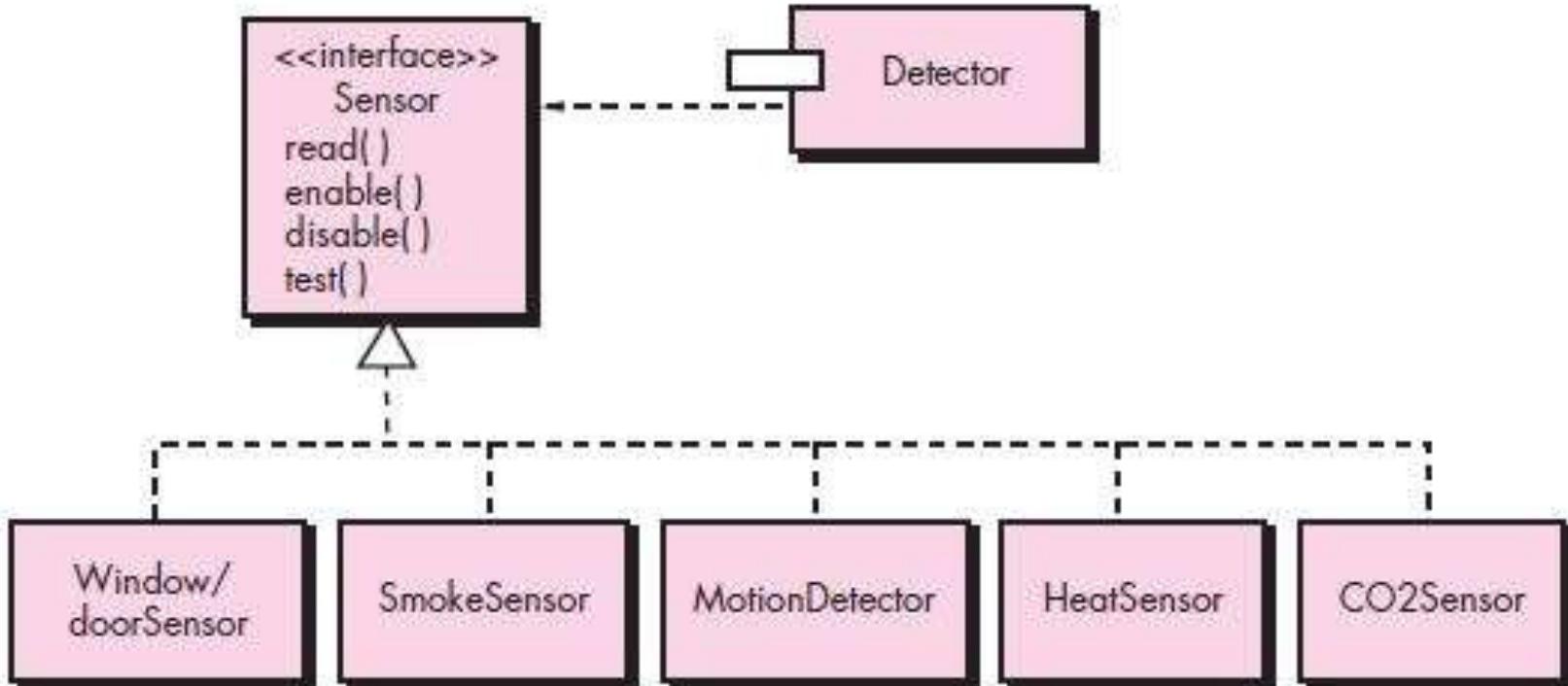
Process-related View

- 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

- 1) Identify all design classes that correspond to the problem domain as defined in the analysis model and architectural model
- 2) 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.
- 3) Elaborate all design classes that are not acquired as reusable components
 - a) Specify message details (i.e., structure) when classes or components collaborate
 - b) Identify appropriate interfaces (e.g., abstract classes) for each component
 - c) Elaborate attributes and define data types and data structures required to implement them (usually in the planned implementation language)
 - d) 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)

- 4) Describe persistent data sources (databases and files) and identify the classes required to manage them
- 5) 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
- 6) 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
- 7) 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

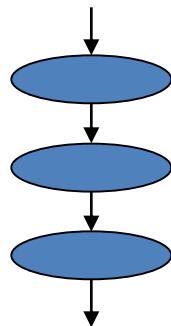
Introduction

- 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**”

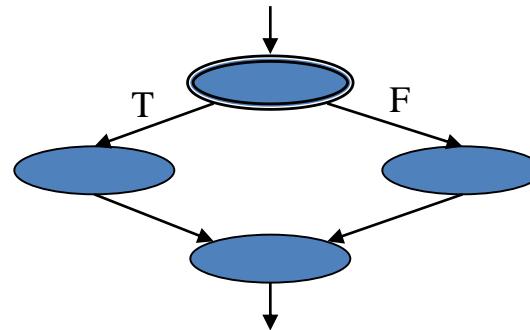
Introduction

- 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

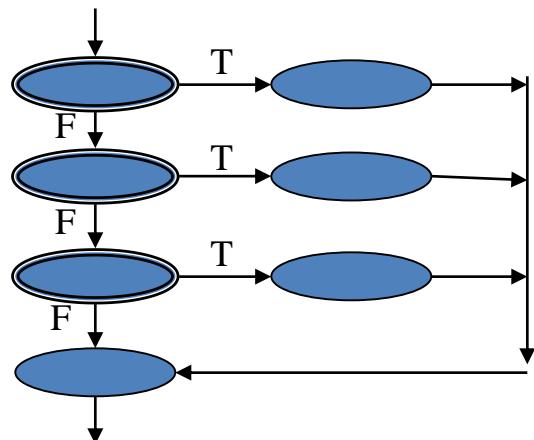
Graphical Design Notation



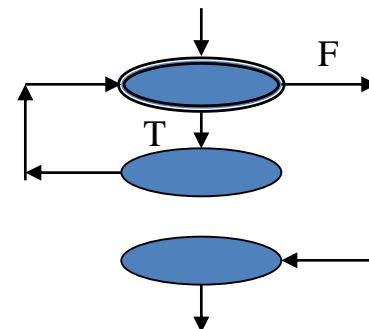
Sequence



If-then-else



Selection



Repetition

Graphical Example used for Algorithm Analysis

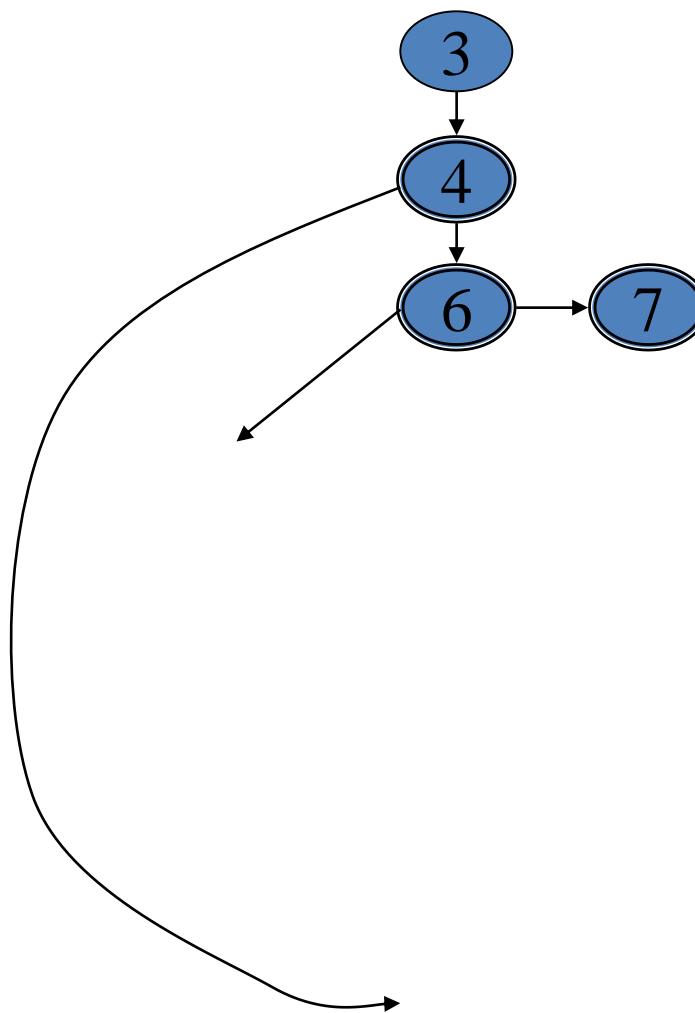
```
1 int functionZ(int y)
2 {
3     int x = 0;
4
5     while (x <= (y * y))
6     {
7         if ((x % 11 == 0) &&
8             (x % y == 0))
9         {
10            printf("%d", x);
11            x++;
12        } // End if
13        else if ((x % 7 == 0) ||
14                  (x % y == 1))
15 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

Graphical Example used for Algorithm Analysis

```
1 int functionZ(int y)
2 {
3     int x = 0;

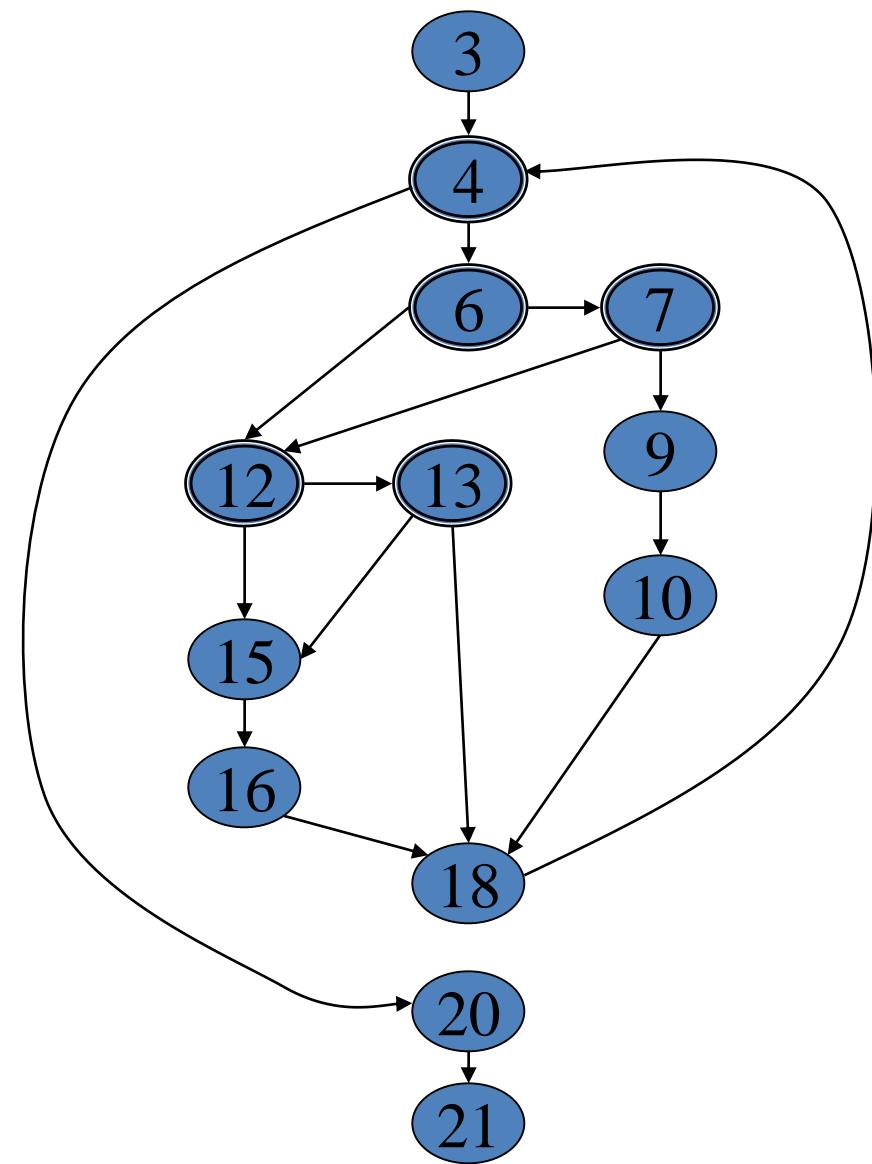
4     while (x <= (y * y))
5     {
6         if ((x % 11 == 0) &&
7             (x % y == 0))
8         {
9             printf("%d", x);
10            x++;
11        } // End if
12        else if ((x % 7 == 0) ||
13                  (x % y == 1))
14        {
15            printf("%d", y);
16        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

20    printf("End of list\n");
21    return 0;
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        printf("\n");
20    } // End while
21
22 printf("End of list\n");
23 return 0;
24 } // End functionZ
```



Tabular Design Notation

- 1) List all actions that can be associated with a specific procedure (or module)
- 2) List all conditions (or decisions made) during execution of the procedure
- 3) Associate specific sets of conditions with specific actions, eliminating impossible combinations of conditions; alternatively, develop every possible permutation of conditions
- 4) 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		✓		✓		✓

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

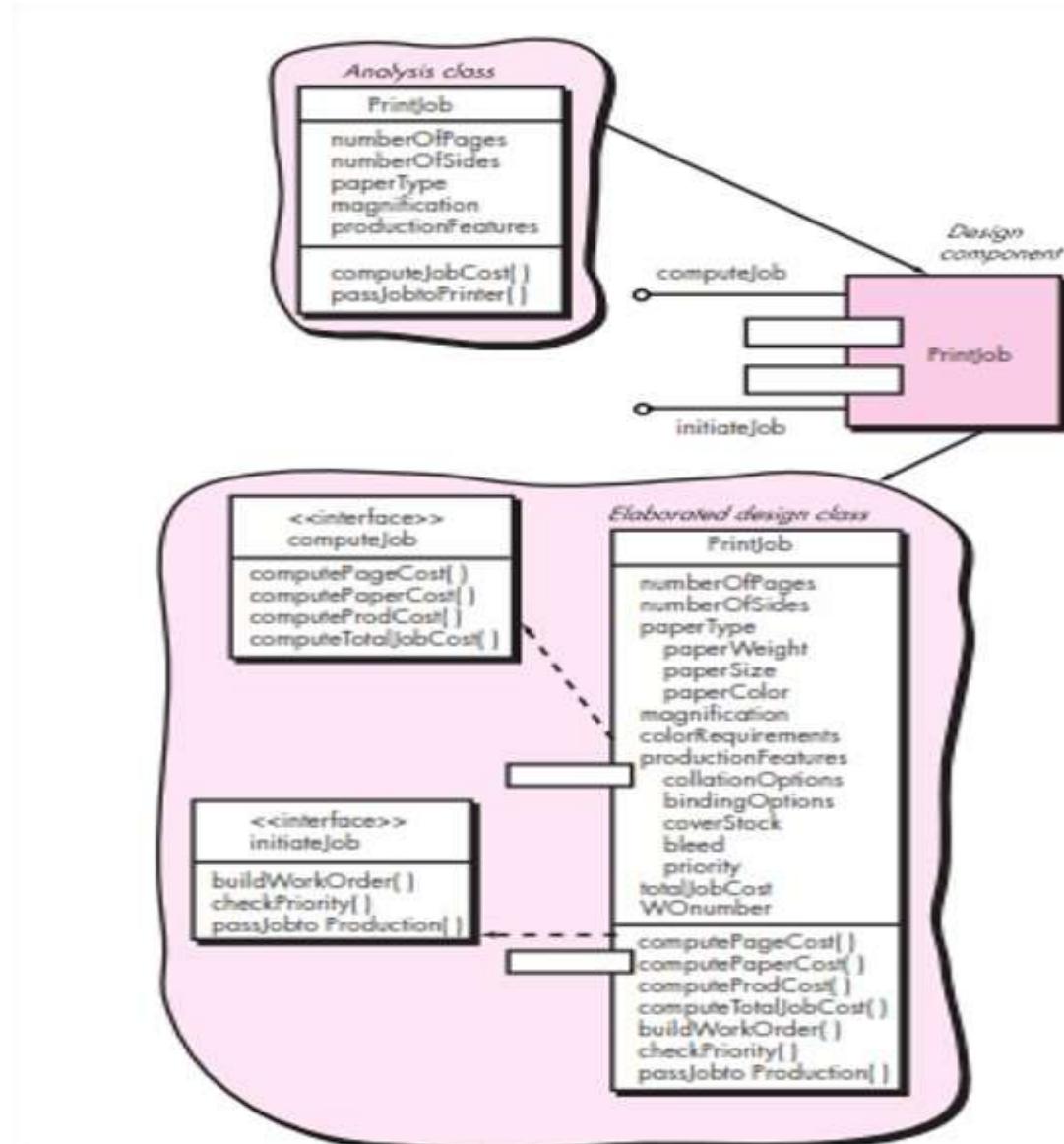
Defined

- 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
 - 1) Provide further elaboration of each attribute, operation, and interface
 - 2) Specify the data structure appropriate for each attribute
 - 3) Design the algorithmic detail required to implement the processing logic associated with each operation
 - 4) Design the mechanisms required to implement the interface to include the messaging that occurs between objects

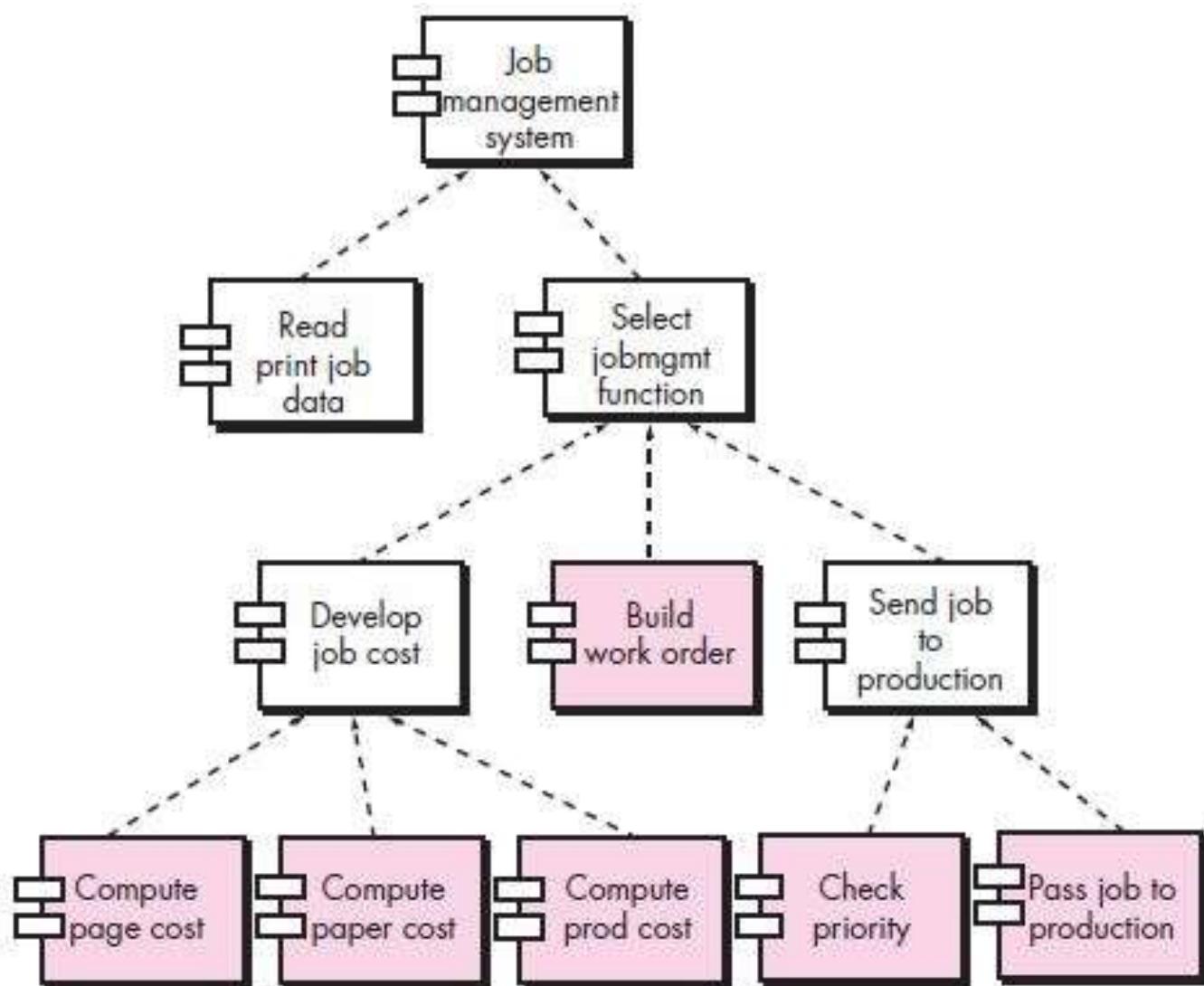
Elaboration of a design component



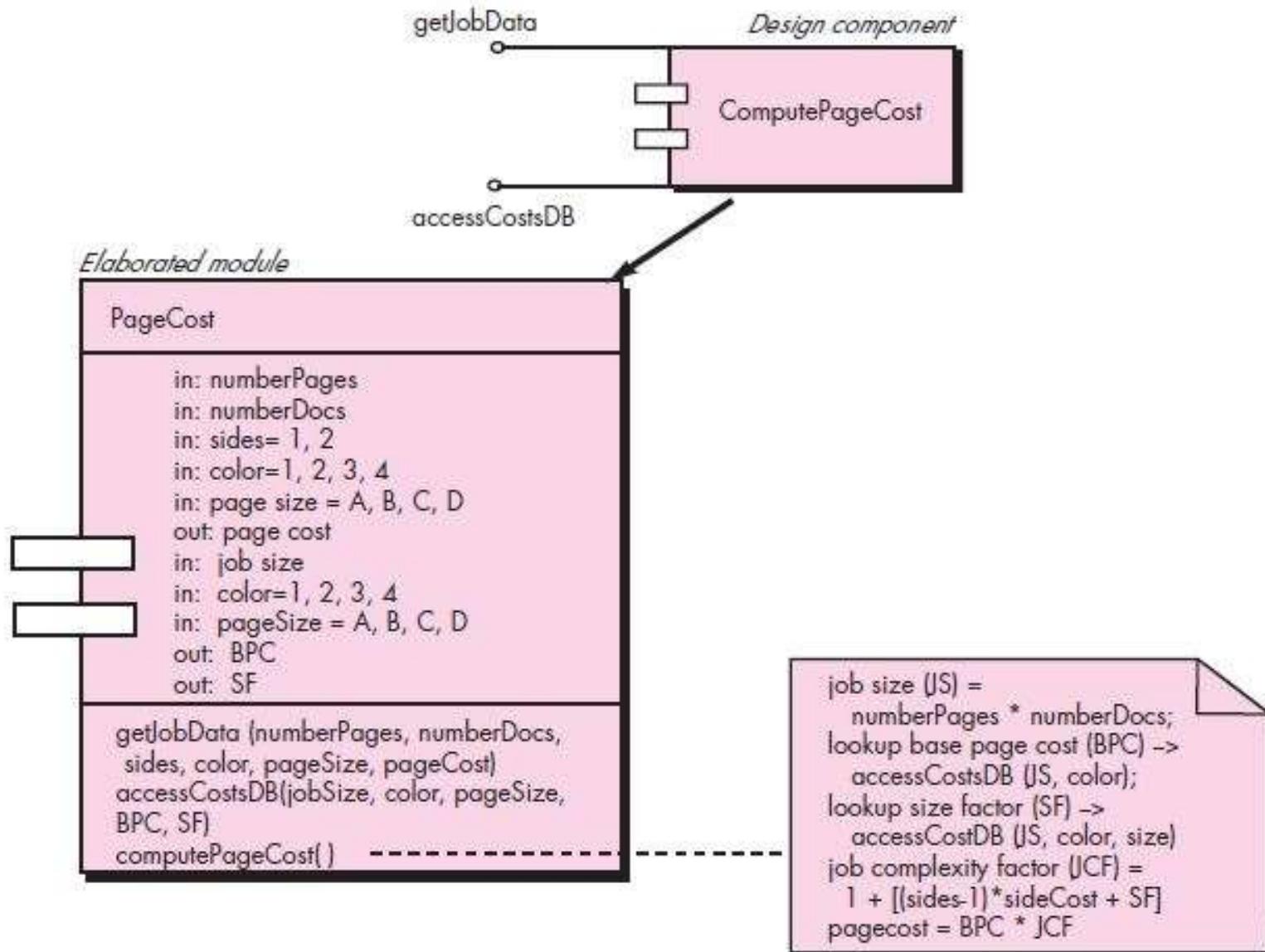
Conventional/Traditional View

- 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
 - 1) Define the interface for the transform (the order, number and types of the parameters)
 - 2) Define the data structures used internally by the transform
 - 3) Design the algorithm used by the transform (using a stepwise refinement approach)

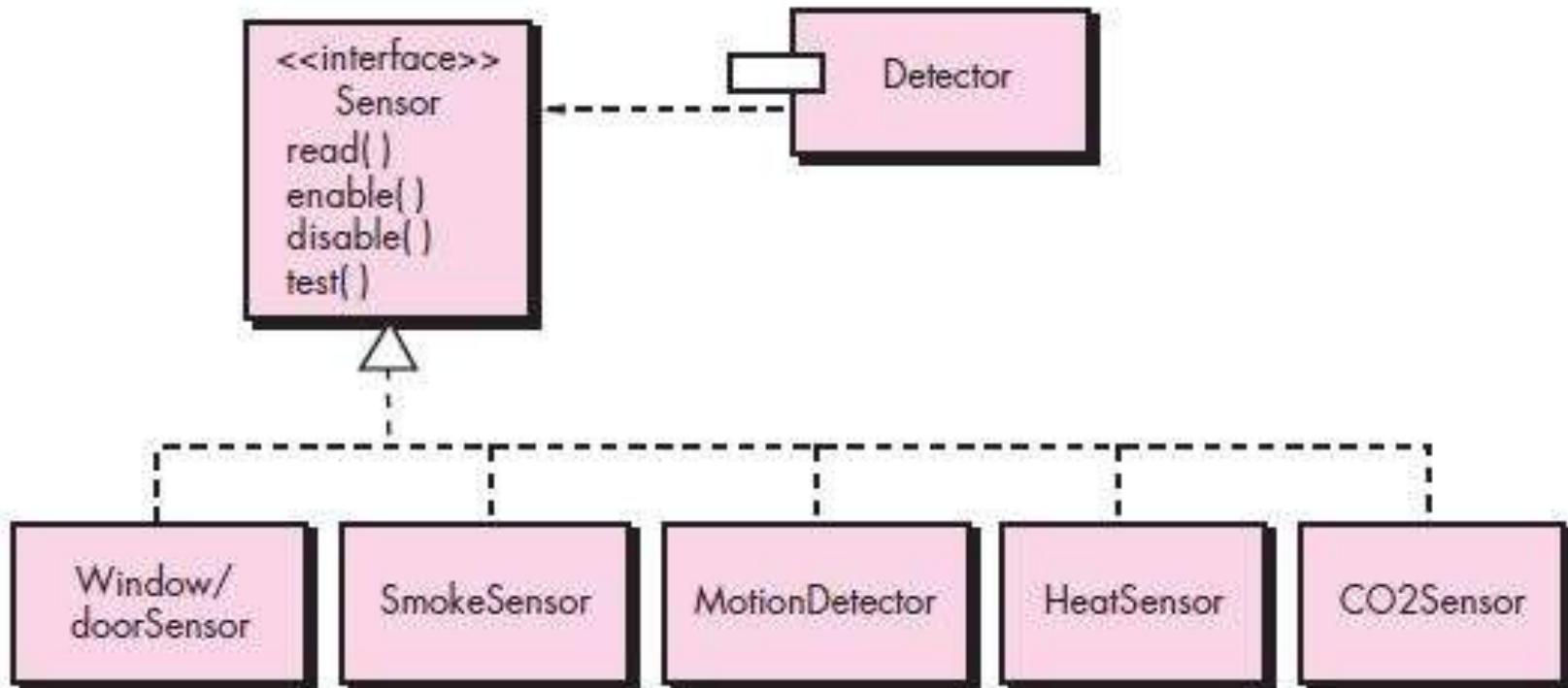
Process-related View

- 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

- 1) Identify all design classes that correspond to the problem domain as defined in the analysis model and architectural model
- 2) 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.
- 3) Elaborate all design classes that are not acquired as reusable components
 - a) Specify message details (i.e., structure) when classes or components collaborate
 - b) Identify appropriate interfaces (e.g., abstract classes) for each component
 - c) Elaborate attributes and define data types and data structures required to implement them (usually in the planned implementation language)
 - d) 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)

- 4) Describe persistent data sources (databases and files) and identify the classes required to manage them
- 5) 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
- 6) 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
- 7) 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

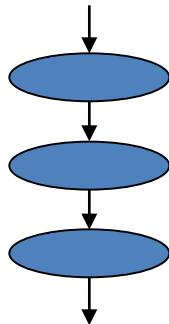
- Various notations depict the use of these constructs
 - Graphical design notation (Control Flow Graphs - CFG)
 - Sequence, if-then-else, selection, repetition (see next slide)
 - Tabular design notation (see upcoming slide)
 - Program Design Language (PDL)
 - Similar to a programming language; however,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

Control Flow Graphs

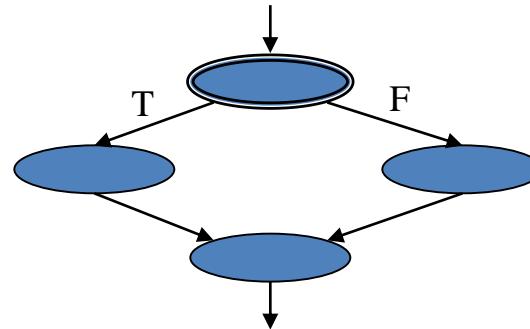
- A Control Flow Graph is a representation of the different blocks of code in a program, and the different paths that the interpreter/compiler can take through the code.
- originally developed by *Frances E. Allen* – American Computer Scientist (*In* 2006 became the first woman to win the Turing Award)
- 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.

Ref: <https://www.teach.cs.toronto.edu/~csc110y/fall/notes/>

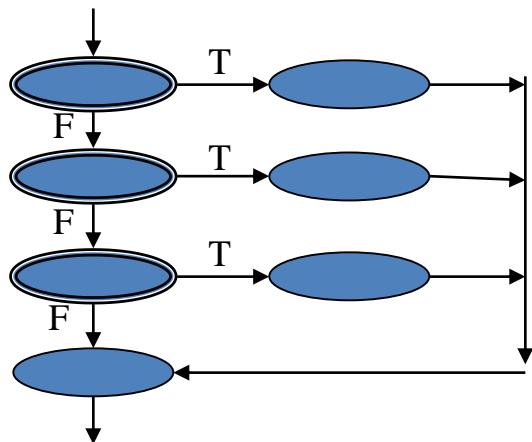
Notations for CFG



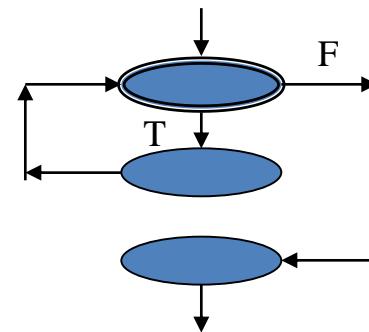
Sequence



If-then-else

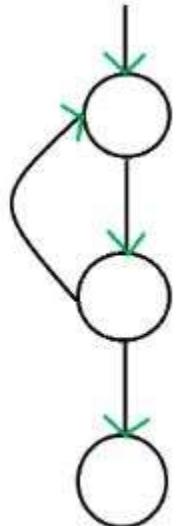


Selection

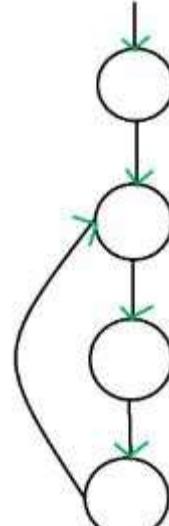


Repetition (While loop)

Notations for CFG contd'



do-while



for

CFG – Example

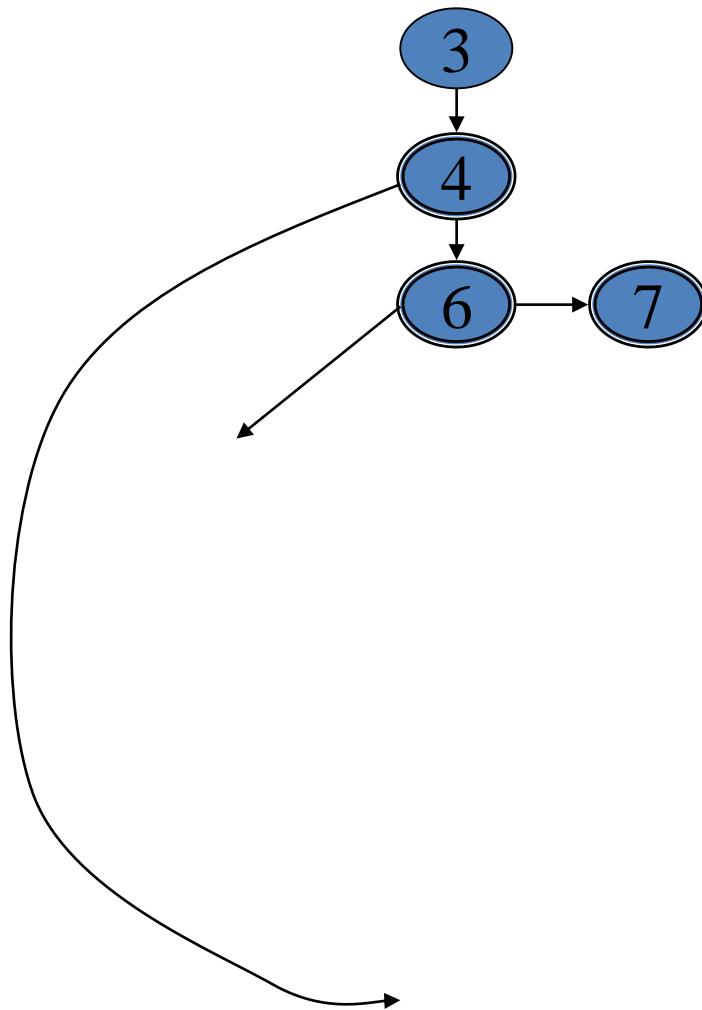
```
1 int functionZ(int y)
2 {
3     int x = 0;
4
5     while (x <= (y * y))
6     {
7         if ((x % 11 == 0) &&
8             (x % y == 0))
9         {
10            printf("%d", x);
11            x++;
12        } // End if
13        else if ((x % 7 == 0) ||
14                  (x % y == 1))
15 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

CFG – Example contd'

```
1 int functionZ(int y)
2 {
3     int x = 0;

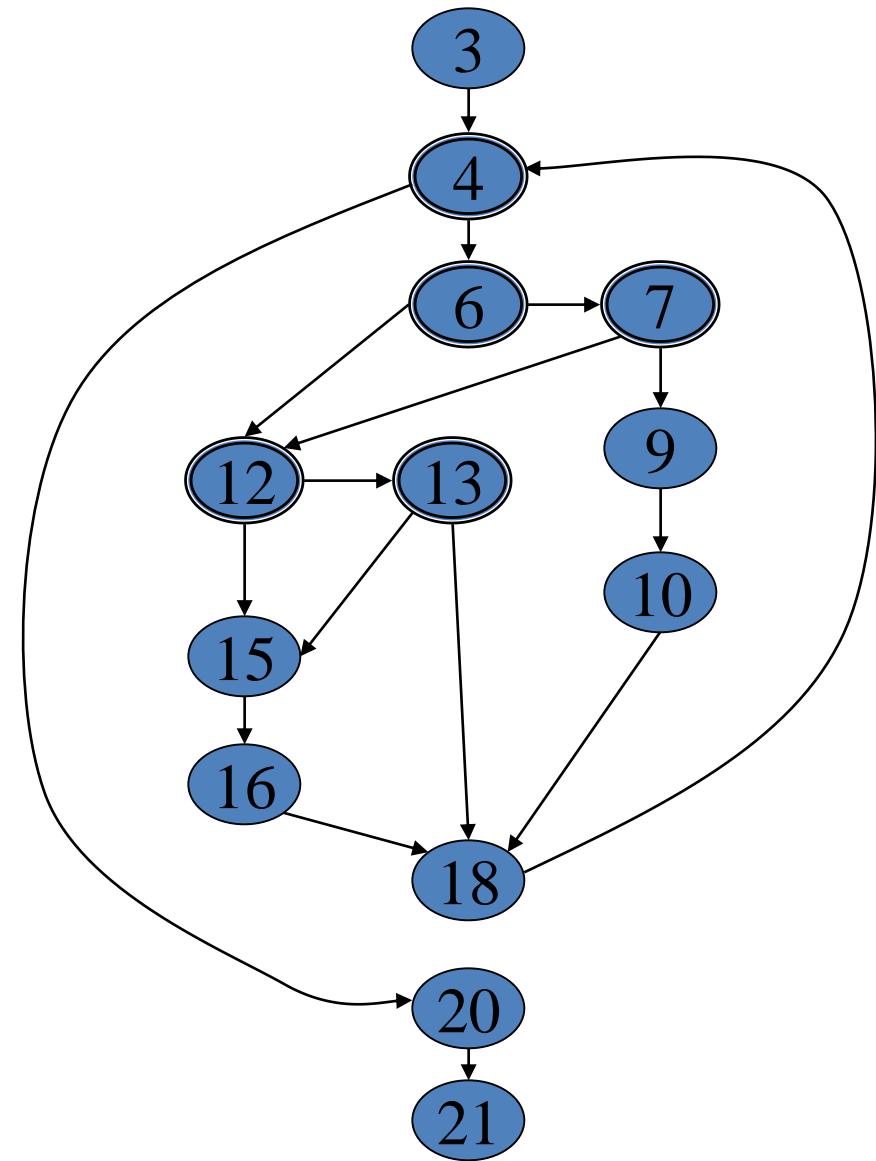
4     while (x <= (y * y))
5     {
6         if ((x % 11 == 0) &&
7             (x % y == 0))
8         {
9             printf("%d", x);
10            x++;
11        } // End if
12        else if ((x % 7 == 0) ||
13                  (x % y == 1))
14        {
15            printf("%d", y);
16        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

20    printf("End of list\n");
21    return 0;
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        printf("\n");
20    } // End while
21
22 printf("End of list\n");
23 return 0;
24 } // End functionZ
```



Tabular Design Notation

- 1) List all actions that can be associated with a specific procedure (or module)
- 2) List all conditions (or decisions made) during execution of the procedure
- 3) Associate specific sets of conditions with specific actions, eliminating impossible combinations of conditions; alternatively, develop every possible permutation of conditions
- 4) 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		✓		✓		✓

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

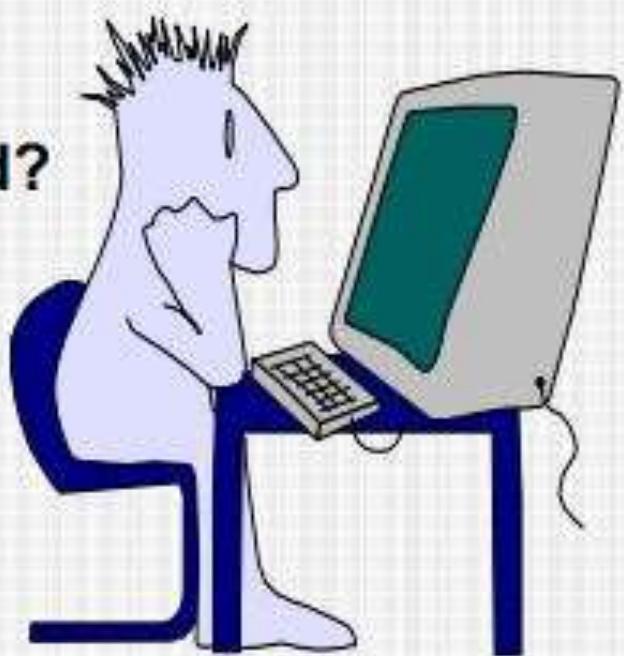
Introduction

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

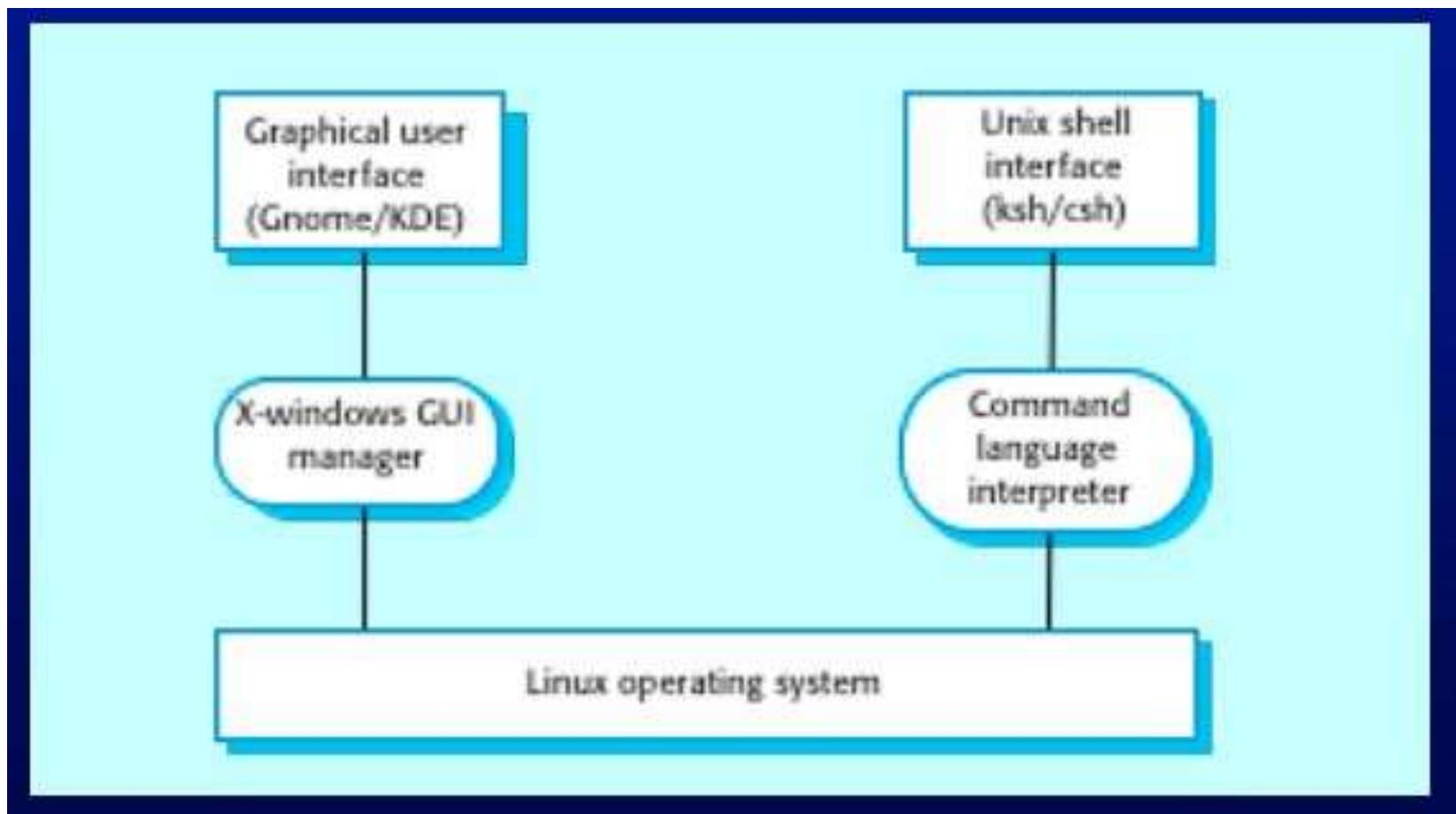
The user interface

- User interfaces should be designed to match the skills, experience and expectations of its anticipated users.
- System users often judge a system by its interface rather than its functionality.
- A poorly designed interface can cause a user to make catastrophic errors.
- Poor user interface design is the reason why so many software systems are never used.

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



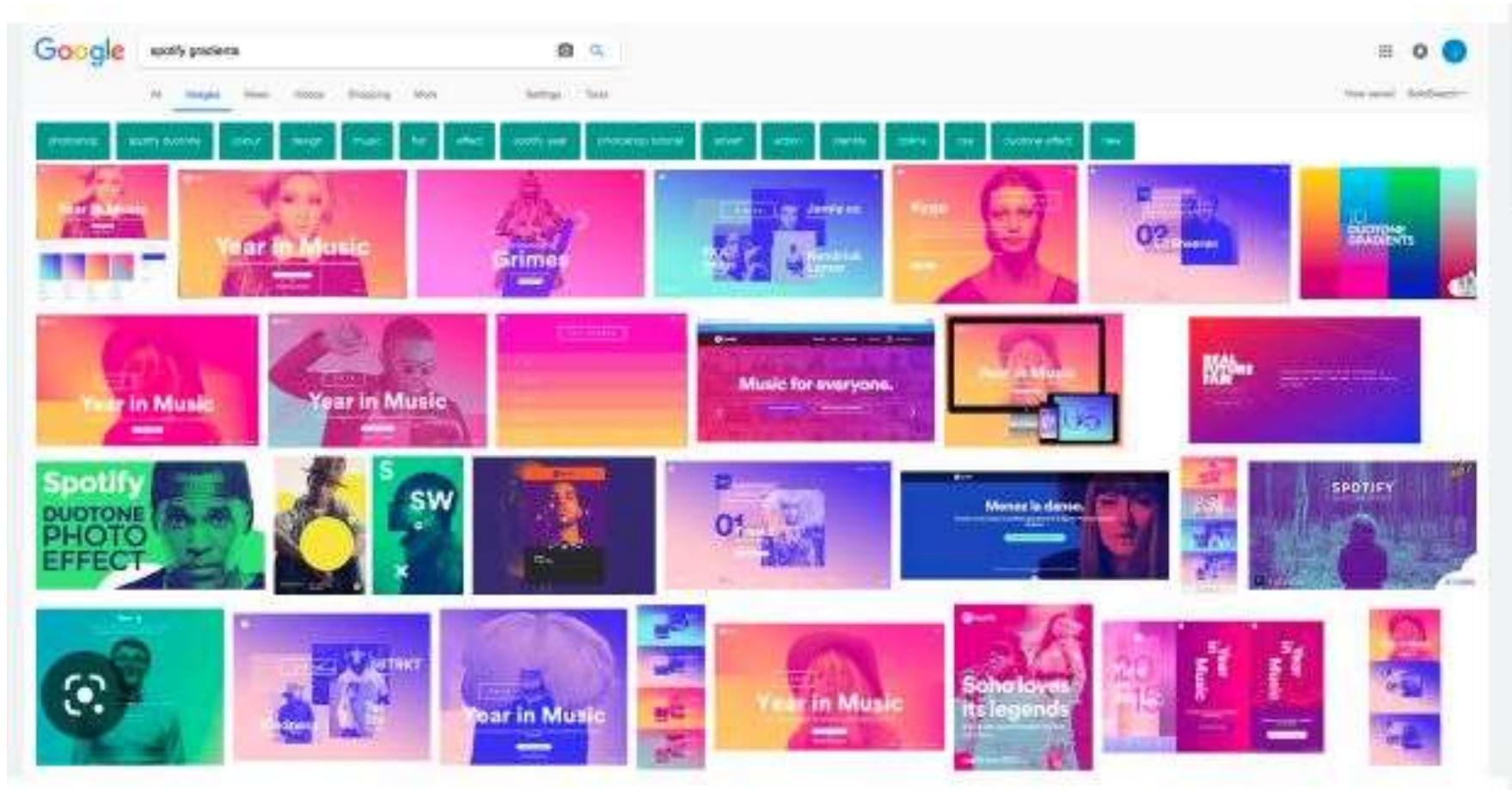
Background

- 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

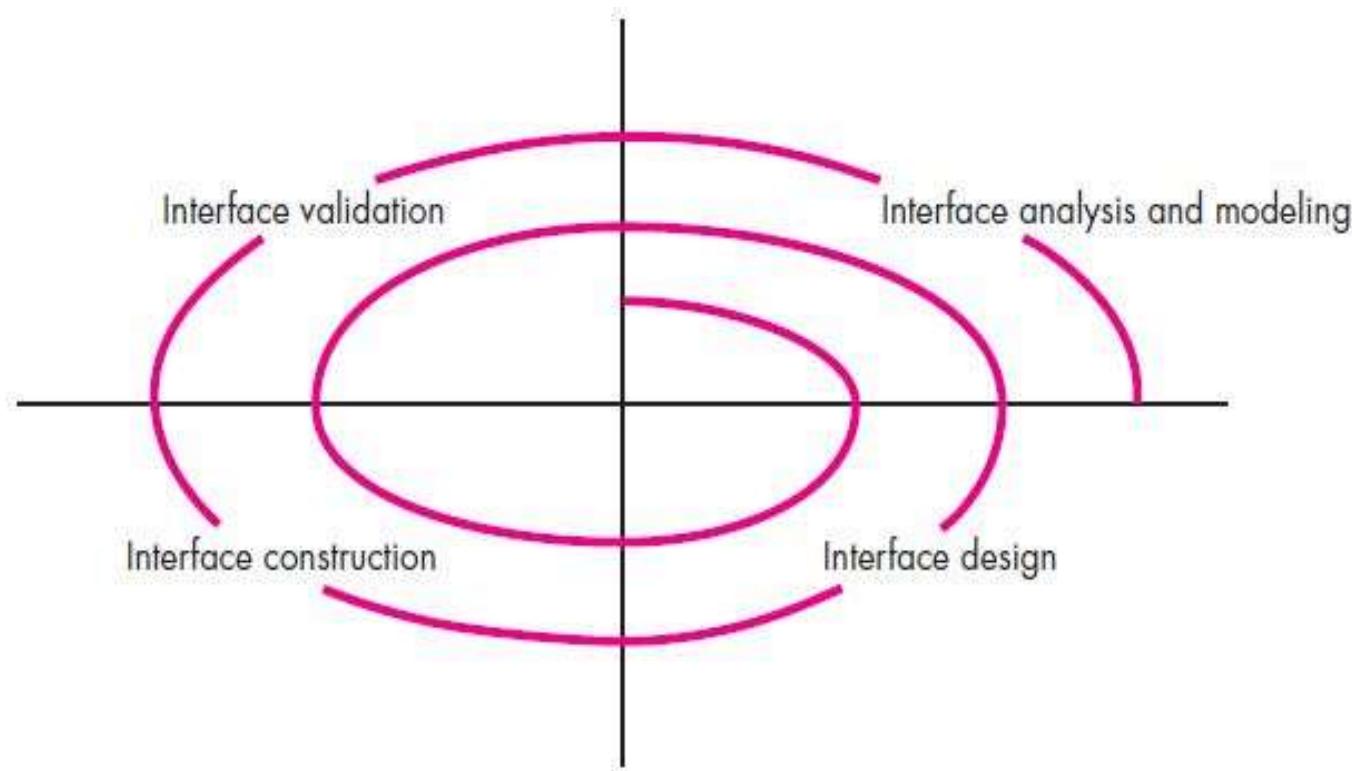
Spotify's color gradient



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 degree to which the interface is easy to use and easy to learn
 - The users' acceptance of the interface as a useful tool in their work

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 "undo" any action

(More on next slide)

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

(More on next slide)

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

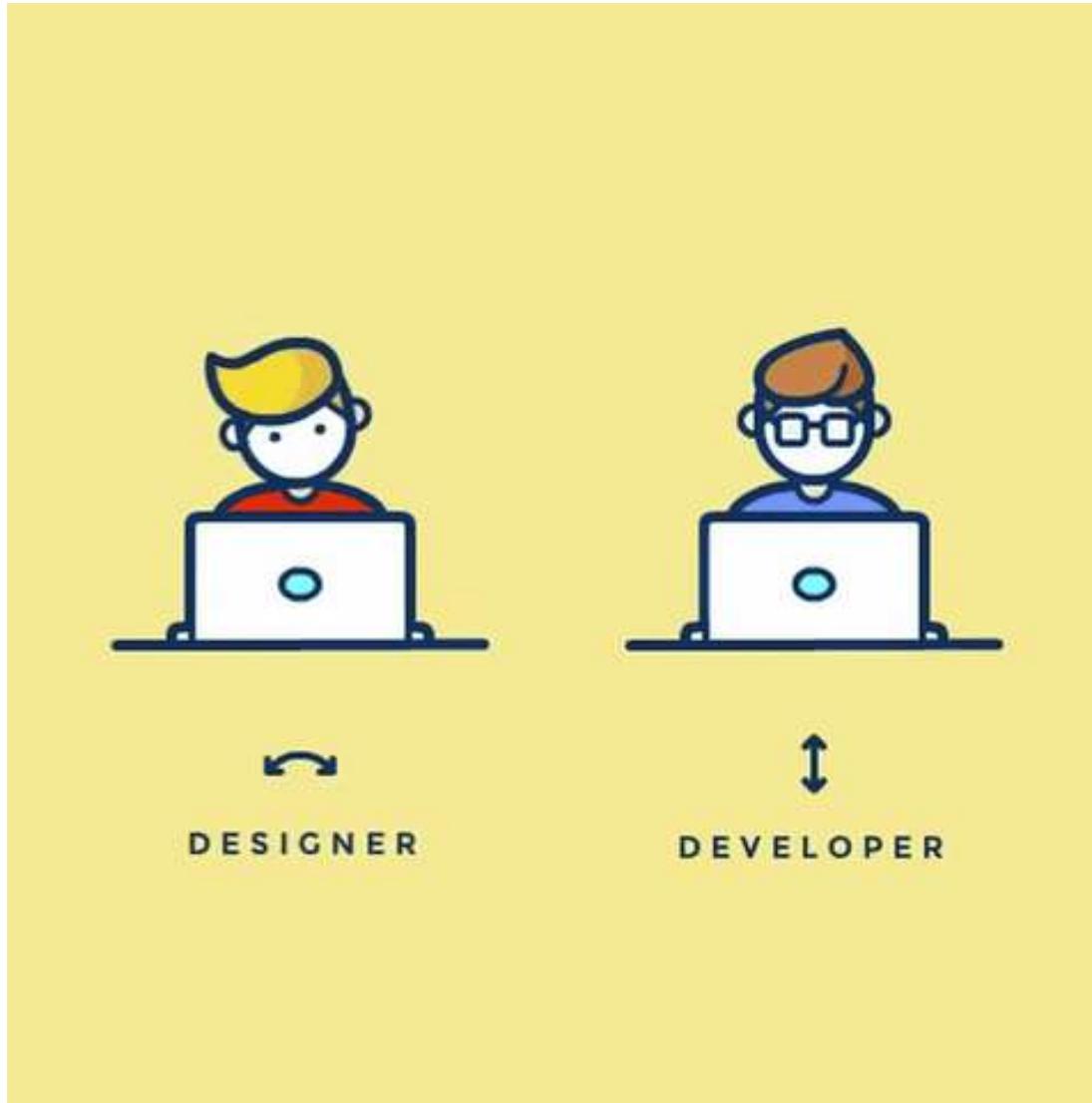
Rule 3. Make the Interface Consistent

- 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 –

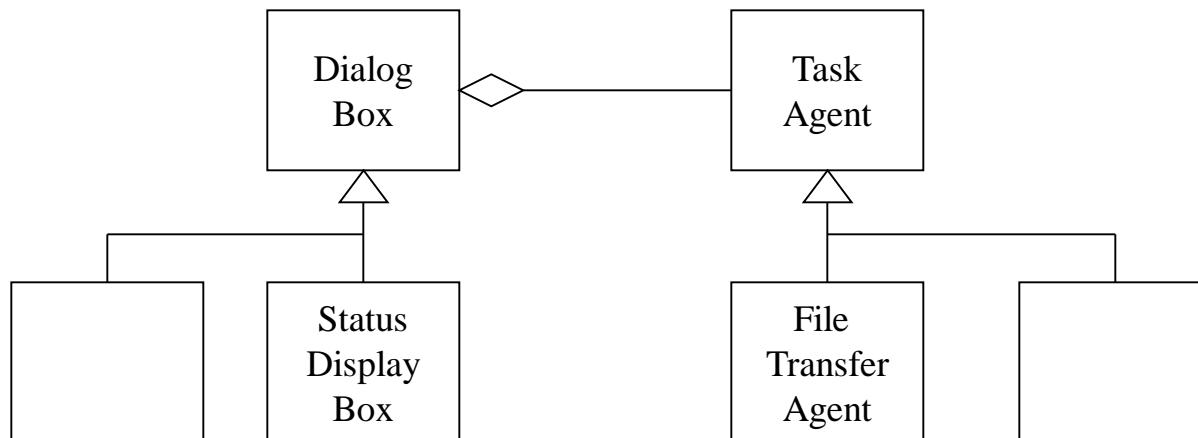
“The truth is, designers and developers—myself included—often think about users. However, in the absence of a strong mental model of specific users, we self-substitute. Self substitution isn’t user centric—it’s self-centric”.

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

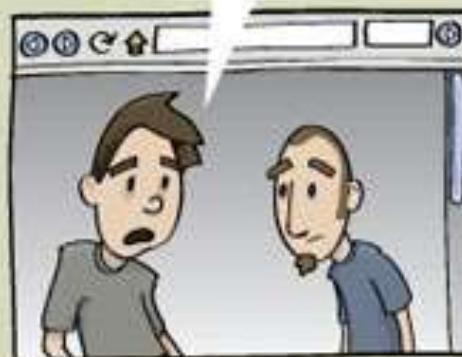
- Often called the [user's system perception](#)
- Consists of the image of the system that users carry in their heads
- Accuracy of the description depends upon the user's profile and overall familiarity with the software in the application domain

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.

MY LEGS! I LOST
MY LEGS!

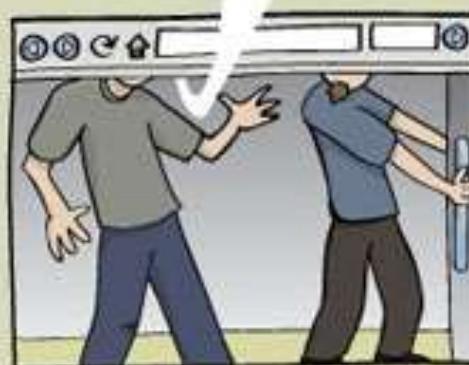
YOU'RE FINE. YOUR LEGS
ARE BELOW THE FOLD.

I KNOW THAT AND
YOU KNOW THAT, BUT
HOW WILL OUR USERS
KNOW THAT?



THEY WILL SCROLL.
THEY DO IT ALL THE TIME.
IT'S NO BIG DEAL.

NO! MY HEAD! I
LOST MY HEAD!



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

- 1) Are the users trained professionals, technicians, clerical or manufacturing workers?
- 2) What level of formal education does the average user have?
- 3) Are the users capable of learning on their own from written materials or have they expressed a desire for classroom training?
- 4) Are the users expert typists or are they keyboard phobic?
- 5) What is the age range of the user community?
- 6) Will the users be represented predominately by one gender?
- 7) How are users compensated for the work they perform or are they volunteers?
- 8) Do users work normal office hours, or do they work whenever the job is required?
- 9) Is the software to be an integral part of the work users do, or will it be used only occasionally?
- 10) What is the primary spoken language among users?
- 11) What are the consequences if a user makes a mistake using the system?
- 12) 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

- 1) Are various types of data assigned to consistent locations on the screen (e.g., photos always in upper right corner)?
- 2) Are users able to customize the screen location for content?
- 3) Is proper on-screen identification assigned to all content?
- 4) Can large reports be partitioned for ease of understanding?
- 5) Are mechanisms available for moving directly to summary information for large collections of data?
- 6) Is graphical output scaled to fit within the bounds of the display device that is used?
- 7) How is color used to enhance understanding?
- 8) 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 elaborates and refines the information developed in the preceding step
- General steps for user interface design
 - 1) Using information developed during user interface analysis, define user interface objects and actions (operations)
 - 2) Define events (user actions) that will cause the state of the user interface to change; model this behavior
 - 3) Depict each interface state as it will actually look to the end user
 - 4) 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
 - Definition of major and minor menu items
 - Specification of a real-world metaphor to follow

Preliminary screen layout

Access Configure System Status View Monitoring

SafeHome

The screenshot displays the SafeHome monitoring software interface. On the left, there's a vertical toolbar with icons for Connect (key icon), Status (key icon), and Video Camera (camera icon). Below the toolbar is a large window titled "Monitoring" showing a floor plan of a house. The floor plan is labeled "First Floor" and includes various sensor and camera locations. Labels include "S door/window sensor", "M motion detector (beam shown)", and "C video camera location". A camera view window titled "Video Image—LR" is also visible, showing a live video feed from a camera located in the Living Room (LR). At the bottom of the interface are control sliders for "Zoom" (ranging from In to Out) and "Pan" (ranging from L to R).

Monitoring

First Floor

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

In Zoom Out

L Pan R

Video Image—LR

SafeHome

Access Configure System Status View Monitoring

Connect

Status

Video Camera

Monitoring

First Floor

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

In Zoom Out

L Pan R

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

Negative Words to Avoid in Your Form Error Messages

Oops

Oops, something wasn't right

Error

 **This Form Has Errors**

Failed

Form submission failed!

Problem

There was a problem creating your account

Invalid

Invalid Fields

Wrong

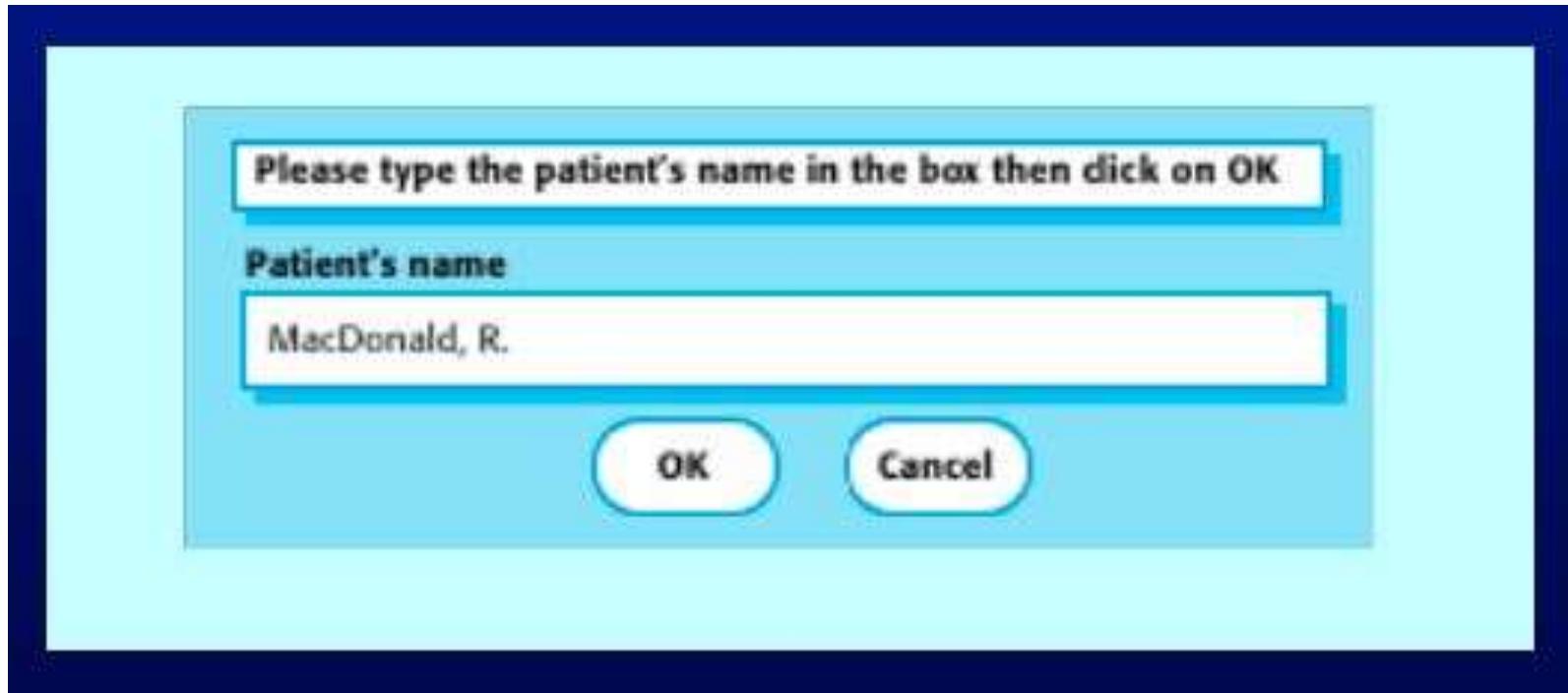
Oops, something has gone wrong

Prohibited

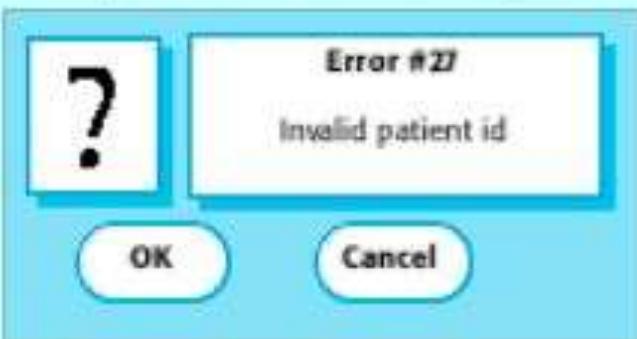
3 errors prohibited this user from being saved

User error

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



Good and bad message design



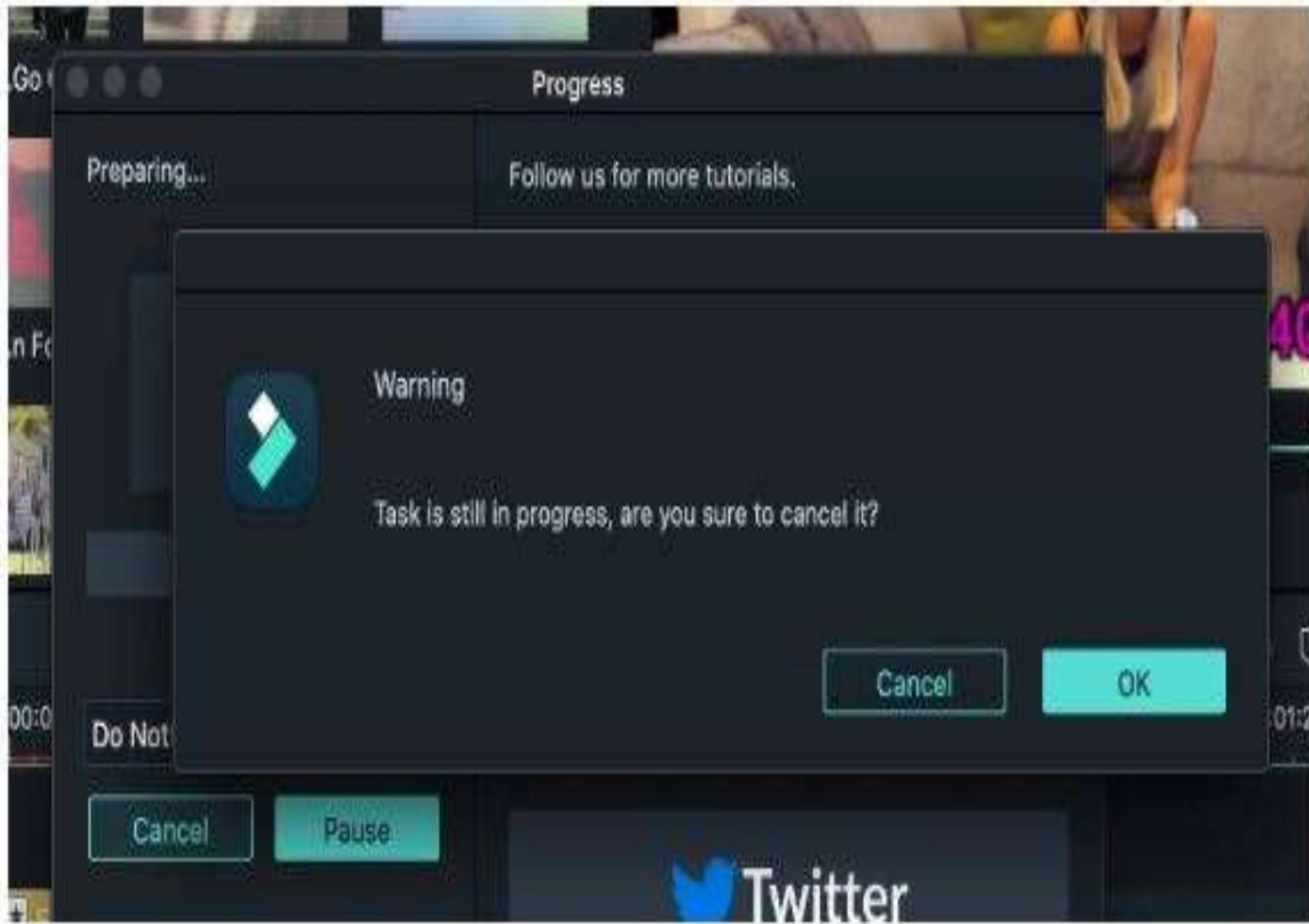
Guidelines for Error Messages

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!

Whoopsy Daizy

xr413r1 error occurred and it's your fault, loser.



I feel like these are trick buttons for users

Action Blocked

Your account has been temporarily blocked from taking this action. Sharing your account with a service that helps you get more likes or followers goes against our Community Guidelines. This block will expire on 2019-07-23. Tell us if you think we made a mistake.

[Tell us](#)

[OK](#)

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

(More on next slide)

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

Webapp Interface design principles and guidelines

Bruce Tognazzi'2001 –

Flexibility

Focus

Fitt's law (*The time to acquire a target is a function of the distance to and size of the target”*)

Human interface objects

Latency reduction

Learnability

Metaphors

Maintain work product integrity

Readability, Track state and Visible navigation

Fitt's law

Index of Difficulty = $\log_2((D/W)+1)$

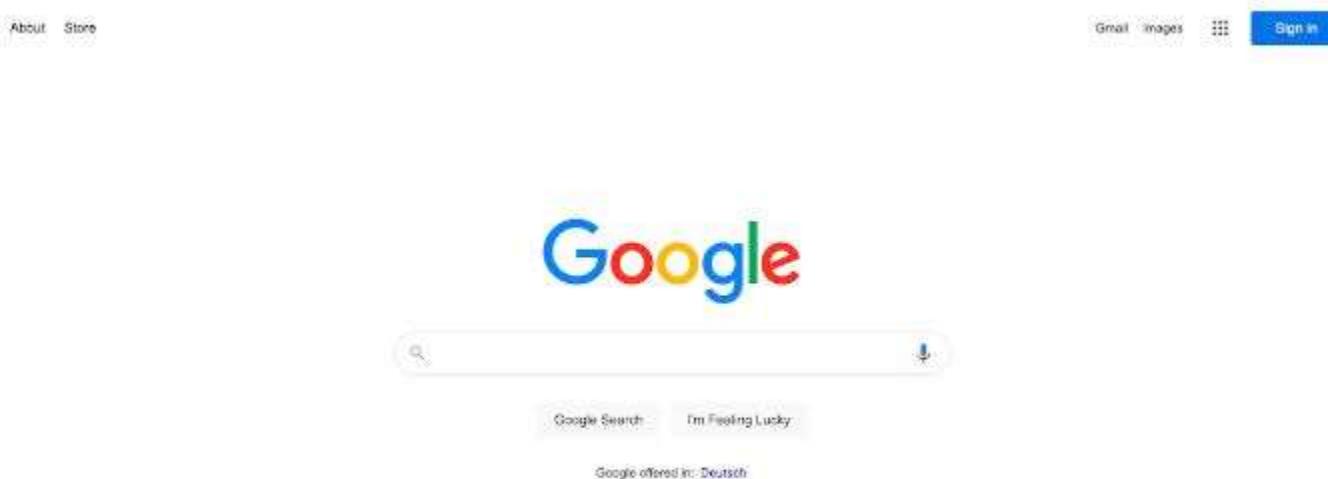
where

- **D** is the distance from the starting point to the target
- **W** is the width of the target along the axis of motion

Some examples of it being used

- A **button** to complete an action that is close to the active elements
- Important elements that are made **larger** so that they're easier to select.
- **Interactive lists** that are as short as possible
- **Menus** are in the top corners of the screen on desktops and the bottom of the screen on mobile

examples of Fitts's Law in UI design



examples of Fitts's Law in UI design

The screenshot shows the Spotify artist profile for Snail Mail. At the top, the artist's name "Snail Mail" is displayed in large white letters against a dark red background featuring a close-up photo of a woman singing. Below the name, it says "690,013 monthly listeners". On the left, there's a green play button icon, a "FOLLOW" button, and a three-dot menu. The "Popular" section lists five songs: 1. Pristine, 2. Heat Wave, 3. Glory, 4. Valentine, and 5. Deep Sea. The song "Pristine" is currently selected. A context menu is open over this song, showing options: "Add to queue" (with a green heart icon), "Go to song radio", "Go to album", "Show credits", "Remove from your Liked Songs" (with a trash bin icon), "Add to playlist", and "Share". To the right of the menu, the song's duration is listed as 4:55. On the far right, there's a "Liked Songs" section with a circular profile picture of the artist and the text "You've liked 2 songs By Snail Mail".

Webapp Interface design principles and guidelines

Bruce Tognazzi'2001 –

Human interface objects

Latency reduction

Learnability

Metaphors

Maintain work product integrity

Readability, Track state and Visible navigation

Nielsen and Wagner's suggestion on interface design guidelines

(Reading speed, signs, scroll, aesthetics, navigation)



References

<https://uxwritinghub.com/error-message-examples/>

<https://ifs.host.cs.st-andrews.ac.uk/Books/SE7/Presentations/PDF/ch16.pdf>

<https://www.springboard.com/blog/design/shareable-ux-design-quotes/>