

CS2SE: Software Engineering



Dr Alexandros Giagkos

School of Computer Science and Digital Technologies
College of Engineering and Physical Sciences

Outline

- ▶ Module introduction
- ▶ Why software engineering?
- ▶ Systems and Modelling
- ▶ UML Component Diagrams

Dr Alexandros Giagkos — a.giagkos@aston.ac.uk

- ▶ Researching bio-inspired optimisation algorithms and their application, networked multi-agent systems, developmental robotics.
- ▶ Teaching object-oriented programming, software engineering, database systems, etc.

Dr Lucy Bastin (Leader) — l.bastin@aston.ac.uk

- ▶ Researching spatial, biodiversity informatics, data quality issues, interoperable standards.
- ▶ Teaching geographic information systems, statistics, remote sensing, etc.

Units 1-6

- ▶ Modelling and Systems
- ▶ Development Process
- ▶ Requirements Engineering
- ▶ Business Analysis
- ▶ OO Systems Analysis
- ▶ Testing

Units 7-12

- ▶ Software Design
- ▶ Architecture
- ▶ OO Detailed Design
- ▶ Design Patterns
- ▶ Implementation
- ▶ Refactoring

Summative Assessment:

100% Computer-based Exam – Open Book (2:30hrs)

- ▶ Software engineering (SE) refers to the systematic procedures used for the analysis, design, implementation, testing and maintenance of software.
- ▶ The term became commonly known in 1968 as the title for NATO's conference to discuss software issues, guidelines, best practices, etc.
- ▶ Coined by NASA's scientist Margaret Hamilton earlier in the 60s.
- ▶ SE's aim is to solve the software crisis.
 - ▶ Failures due to increasing demands & low expectations.

Software engineering sets apart a hobbyist programmer from a professional software developer.

- ▶ It is the standardised, structured and thorough approach to writing code.
- ▶ Treats the whole development process from start to finish in a formalised manner.
- ▶ It allows us to deliver programs that meet a certain level of rigour (i.e., it is correct, efficient and secure).

Systems thinking is the **holistic approach** to solving problems that focuses on how a system's **components interrelate and change over time**, within the context of the **overall missions** the system is designed to accomplish.



A bicycle is a system.

- a) yes
- b) no
- c) maybe

A pile of papers is a system.

- a) yes
- b) no
- c) maybe

A tree is a system.

- a) yes
- b) no
- c) maybe

System Characteristics

Systems have the following characteristics:

- ▶ **Central objective**

All components work together to achieve common goal.

- ▶ **Integration**

Components are put together to complete a system.

- ▶ **Interaction**

The system works as a result of components interacting with each other.

- ▶ **Interdependence**

A change to one component will affect other components.

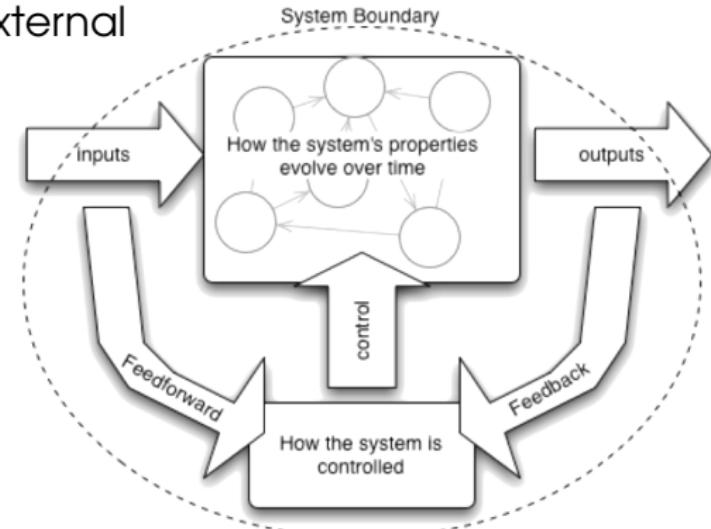
- ▶ **Organisation (hierarchical)**

The right components in the right place, in the right time.

Expected Components in a System

Systems have the following types of components:

- ▶ Input, processes and output
- ▶ Environment: internal & external
- ▶ Boundary
- ▶ Interface
- ▶ Feedback and control



Hierarchy in Systems

- ▶ Complex systems are comprised of various subsystems.
- ▶ Subsystems are arranged in hierarchies and are integrated to achieve the common goal.
- ▶ Subsystems have their own boundaries, thus interfaces and environment.



Online shopping UML component diagram example with three related subsystems - WebStore, Warehouses, and Accounting.

- ▶ Users and/or other systems interact with a system via interface components.
- ▶ The external (operational) environment affects the functioning of a system.
- ▶ A system's function may be to change its environment.
- ▶ External environment can be both physical and organisational.

The complex relationships between the components in a system mean that a system is more than simply the sum of its parts.

Properties emerge once the system components are integrated.

There are two types of emergent properties:

- ▶ Functional emerge when all the parts of a system work together to achieve some objective.
- ▶ Non-functional relate to the behaviour of the system in its operational environment.

The motorbike weights 180 kg.

- a) emergent property
- b) non-emergent property

The motorbike can be ridden.

- a) emergent property
- b) non-emergent property

The lights of the bike can be turned on.

- a) functional emergent property
- b) non-functional emergent property

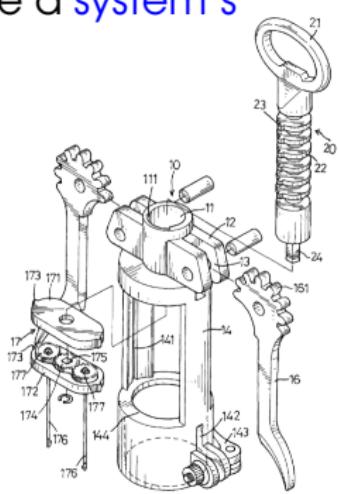
The ATM provides a secure service.

- a) functional emergent property
- b) non-functional emergent property

The world is full of **systems** and **sets**, with the majority of people seeing only the latter.

What is required to propose or improve an existing system?

- ▶ Use our "*component-based vision*" to create a **system's model** out of a system.
- ▶ Model it, no matter how crude the overall insight of it might be.



A system's model must have *the right level of detail* to define:

- ▶ The components that constitute the system.
- ▶ The types of relations between those components.
- ▶ How the components interrelate into the whole system to perform some function.
- ▶ How the whole system interacts with its environment.

Why are models useful?

Models are useful for:

- ▶ Abstraction

Allows the modeller to focus on the most important features of a real-world situation.

- ▶ Representation

Makes one's ideas into a more concrete artefact.

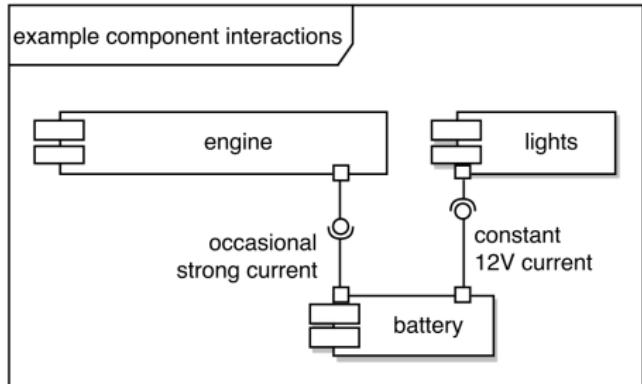
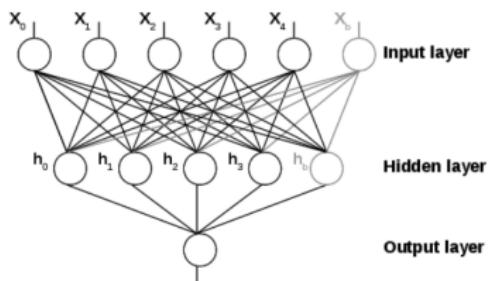
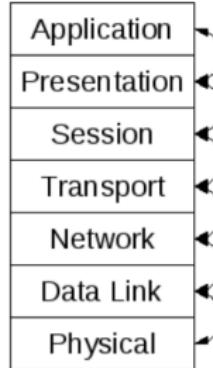
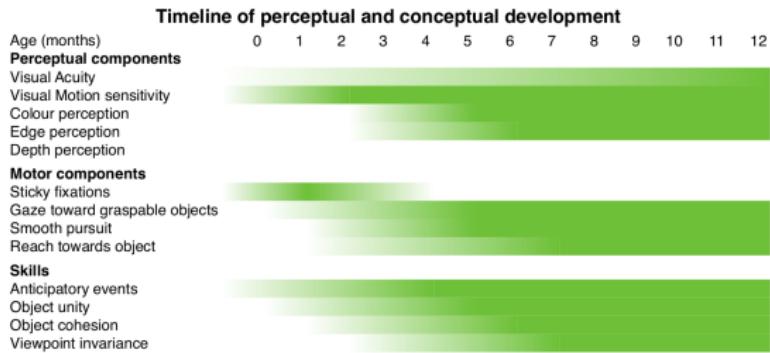
- ▶ Communication

Help sharing ideas and thoughts that are difficult to put into words.

- ▶ Early evaluation

No runtime failures, check if requirements are met.

Examples of Models



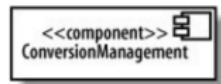
Advantages of a model over the modelled system:

- ▶ A model is quicker and cheaper to build.
- ▶ One can choose which details to include or omit in a model — easier to analyse.
- ▶ Some models can be used in simulated environments — cheaper and safer option.
- ▶ Models evolve (no need to be perfect!) and in some cases they do not even need to make sense.

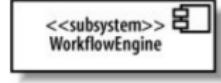
- ▶ Software is an abstract entity — its models are abstract too!
- ▶ Program code = detailed model?
- ▶ Models of software focus on certain aspects:
 - ▶ Code organisation.
 - ▶ Runtime entities and their responsibilities.
 - ▶ Overall data flow.
 - ▶ Messages among entities.
- ▶ **Unified Modelling Language** (UML) provides all the tools necessary to produce models.

Component diagrams help us plan out the high-level pieces of a system to establish the architecture and manage complexity and dependencies amongst components.

Component:



Subsystem:



Component providers:

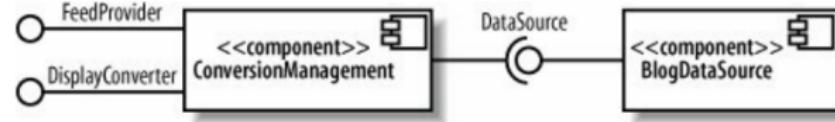


Two common ways to depict how components work together.

Using an arrow for the dependency:

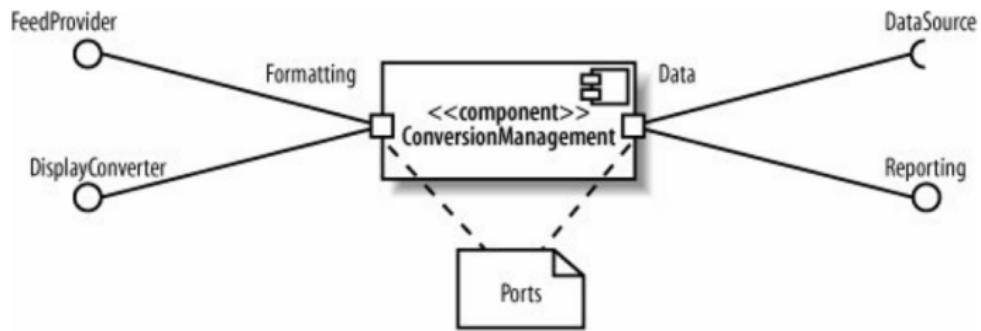


Ball and socket connection:



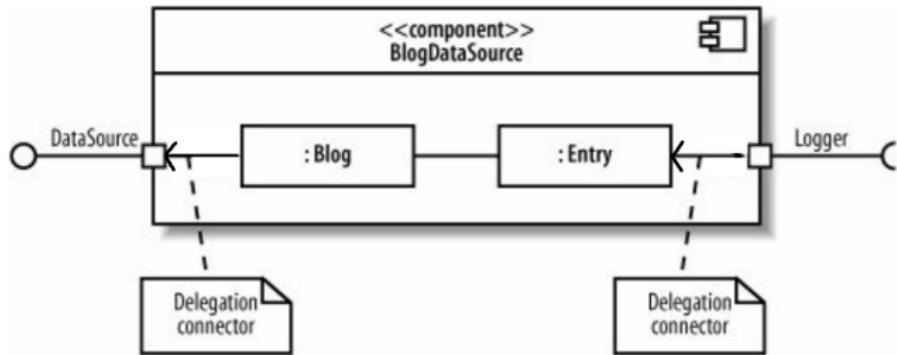
Ports are used to depict points of interaction between a component and the outside world.

Ports usually group interfaces together:



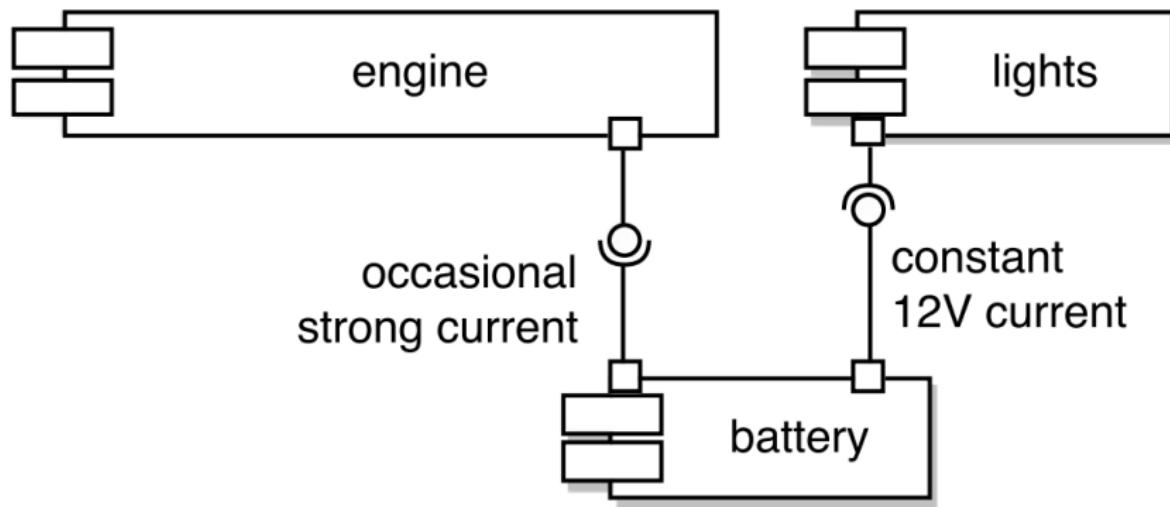
One can depict the **internal structure** of a component and use **delegation connectors** to show the *direction of traffic*:

Ports usually group interfaces together:



Useful Notation for Component Diagrams: The Circuit Example

example component interactions



Today's learning outcomes:

- ▶ You were introduced to the module tutors, structure and ways of assessment.
- ▶ We defined software engineering and discussed about its importance.
- ▶ You were introduced to systems thinking and systems modelling.
- ▶ We discussed UML Component Diagrams and their notation.

CS2SE: Software Engineering



Dr Alexandros Giagkos

School of Computer Science and Digital Technologies
College of Engineering and Physical Sciences

Q: How.. ?

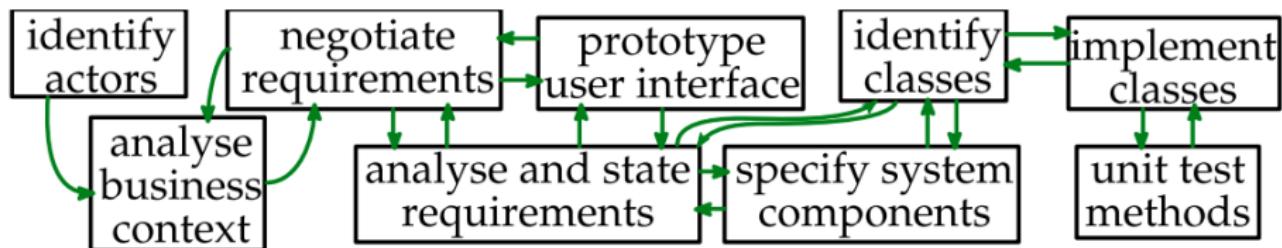
By considering..

- ▶ Software Development Lifecycle Processes
- ▶ UML Activity Diagrams

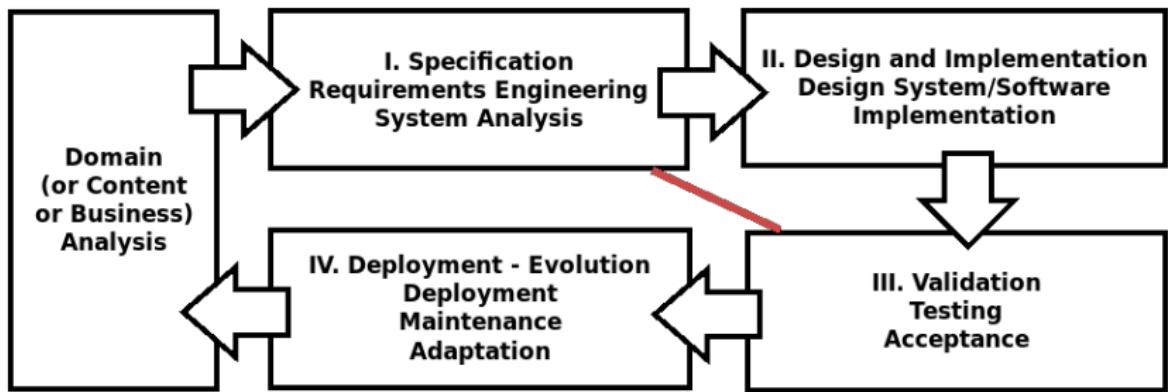
Software Development Lifecycle

- ▶ Software Development Lifecycle (SDLC) or *software development process/methodology* is a sequence of project phases we can follow in order to create a software application.
- ▶ Several models and frameworks from the early 60s until today.
- ▶ SDLC =
activities required to develop software AND
an approach to linking them together in a work flow.

- The need for a logical progression of development activities



- Logical progression, not necessarily time progression.



What do we need to do to develop an “intelligent cookbook app”? Assign activities to their category.

Business context

DB schemas

Specification

observe chefs working

Design+Implementation

update user stories

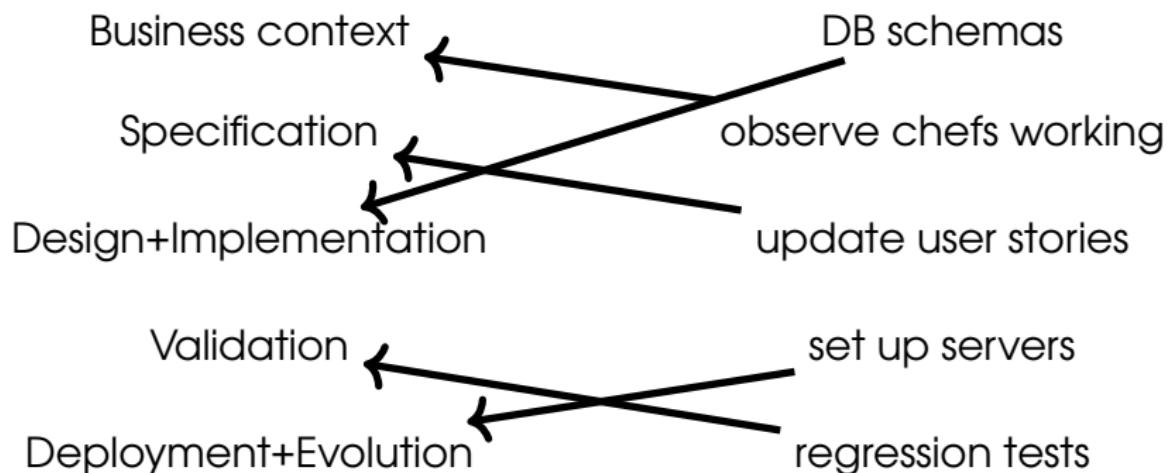
Validation

set up servers

Deployment+Evolution

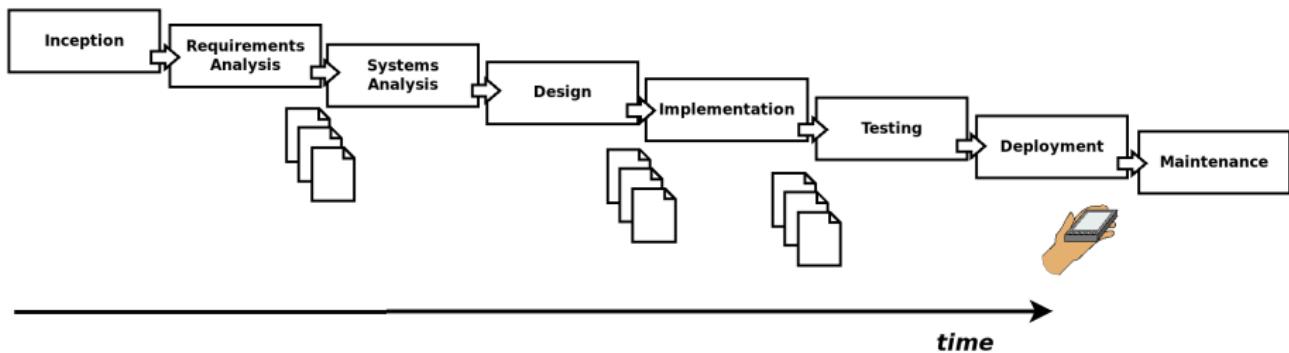
regression tests

What do we need to do to develop an “intelligent cookbook app”? Assign activities to their category.



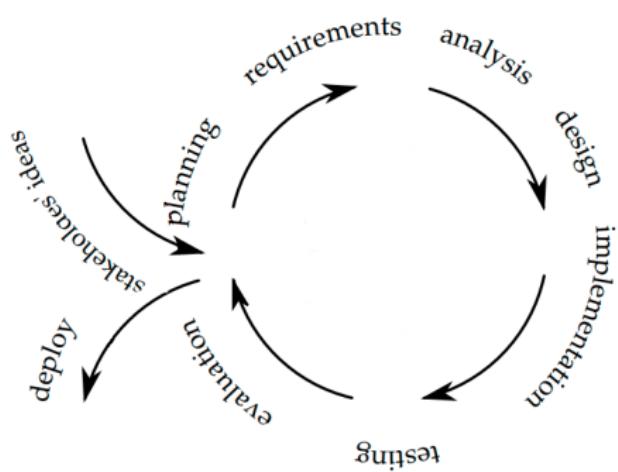
- ▶ **Domain analysis:** study what is already there, literature review, conversations with experts, etc.
- ▶ **Specification (Requirements Engineering):** feasibility study, user requirements elicitation and analysis (input from stakeholders), systems analysis, etc.
- ▶ **Design:** model software structures, software prototypes, interface specifications, component models, etc.
- ▶ **Implementation:** source code, database, interfaces, data models, deployment plan, etc.
- ▶ **Validation:** component testing (unit testing, module testing), integration testing (sub-system and system testing), user testing (acceptance testing), etc.
- ▶ **Deployment:** deployed system, user manuals, training documentation, training staff, fault report, etc.

- ▶ The simplest way of organising the development – very rarely used these days.
- ▶ Large documents and collections of models created at most stages.
- ▶ In the classical model there is no turning back (some modern variants allow backtracking).

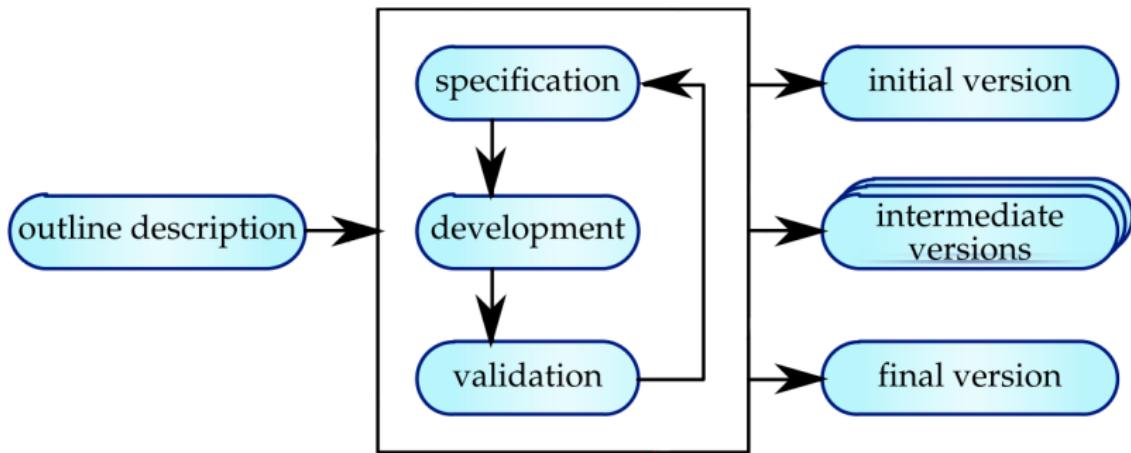


Cyclic: Iterative/Incremental

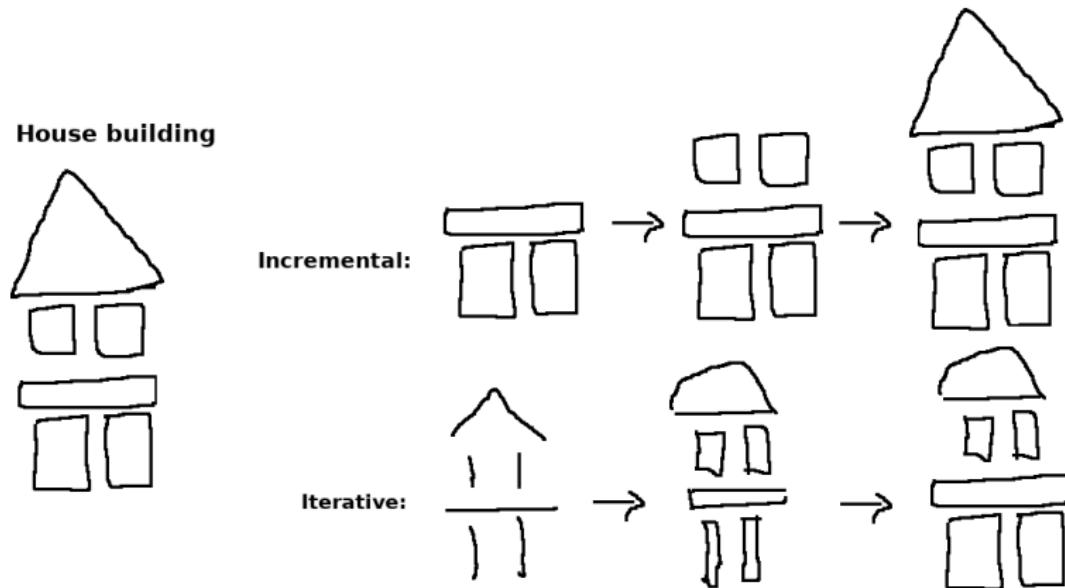
- ▶ Cyclically repeating (a subset of) waterfall activities.
- ▶ A cycle adds a feature or makes improvements.
- ▶ Typically cycles are 1-4 weeks short.
- ▶ You can, if needed, deploy stakeholder's ideas.



- ▶ Cycle: a sequence of developing and evolving prototypes.
- ▶ From some point on, products are deployable.

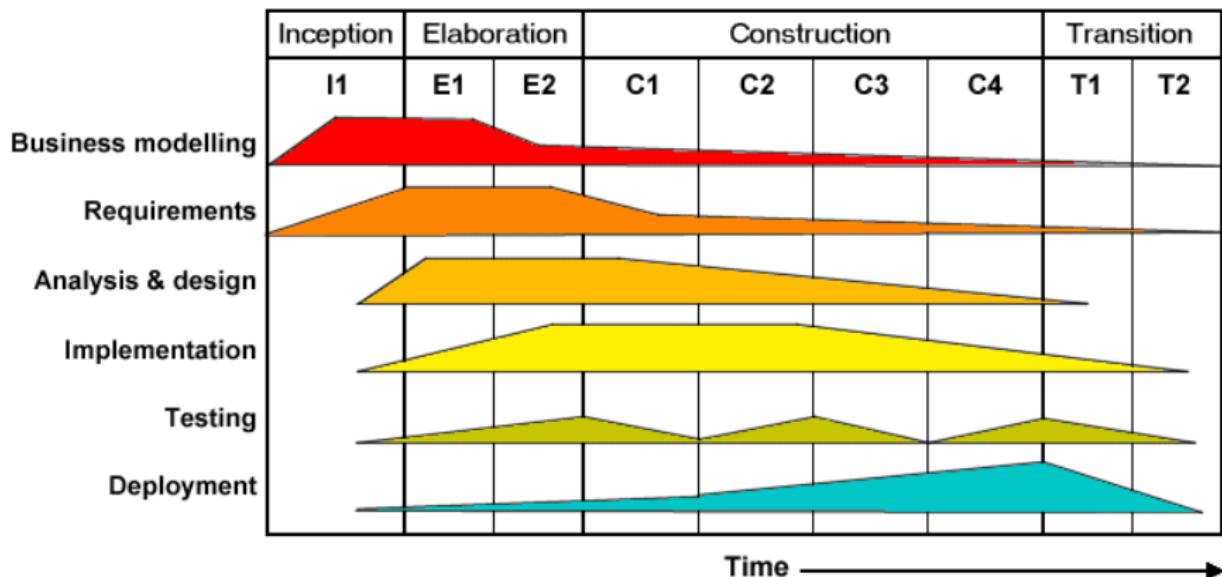


- ▶ **Incremental**: adding new features in each version.
- ▶ **Iterative**: making improvements with each cycle.



- ▶ The **Unified Process** is iterative:
 - ▶ **Inception phase:**
project scope, communication with stakeholders, goals, planning, costs → vision document, use case models, etc.
 - ▶ **Elaboration phase:**
analysing problem domain, eliminating major risks, refining plan → actors identified, clarifying requirements, etc.
 - ▶ **Construction phase:**
delivering the “beta” version, meeting all stakeholders’ needs, testing → a working product ready to be deployed, test results, documentation, etc.
 - ▶ **Transition phase:**
deployment and maintenance of the system, bug reports, training users → project completed and in use, debriefing.

- Performing SDLC activities to different extent:



- ▶ Agile is an approach to software development.
- ▶ Also, a movement, a mindset.

“We do not act rightly because we have virtue or excellence, but we rather have those because we have acted rightly. We are what we repeatedly do. Excellence, then, is not an act but a habit.”

— Aristotle, Nicomachean Ethics

- ▶ Agile was another answer to software crisis in the '70s.

"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 terms on the right, we value the items on the left more."

<http://agilemanifesto.org/>

- ▶ Face-to-face communication among team and customers.
- ▶ Customer representative always available.
- ▶ Software developed in quick iterations and tested continuously.
- ▶ Good quality, well documented code.
- ▶ No long-term planning, but setting long-term goals and visions.
- ▶ Simple design to meet personal needs, refactor due to changes.

Scrum adds:

- ▶ Communication via several types of regular meetings.
- ▶ Stakeholders have to trust developers.
- ▶ Prioritisation of features via backlogs (i.e., todo lists).

eXtreme Programming adds:

- ▶ Pair programming.
- ▶ Sustainable pace, no overtime.
- ▶ Shared system metaphors.
- ▶ Shared code ownership – anyone can edit.



- ▶ Roots in a 1986 paper by Takeuchi and Nonaka, "The new new product development game".
- ▶ Developed in 1990, formalised by Sutherland and Schwaber in 1995.
- ▶ def **Scrum** (n): A framework within which people can address complex adaptive problems, while productively and creatively delivering products of the highest possible value.

Pillars

- ▶ Transparency: everybody involved in delivering the project should understand what is expected and how the process works.
- ▶ Inspection: review the progress at regular intervals.
- ▶ Adaptation: change a process if it is not working, or is not delivering a sensible result.

Values

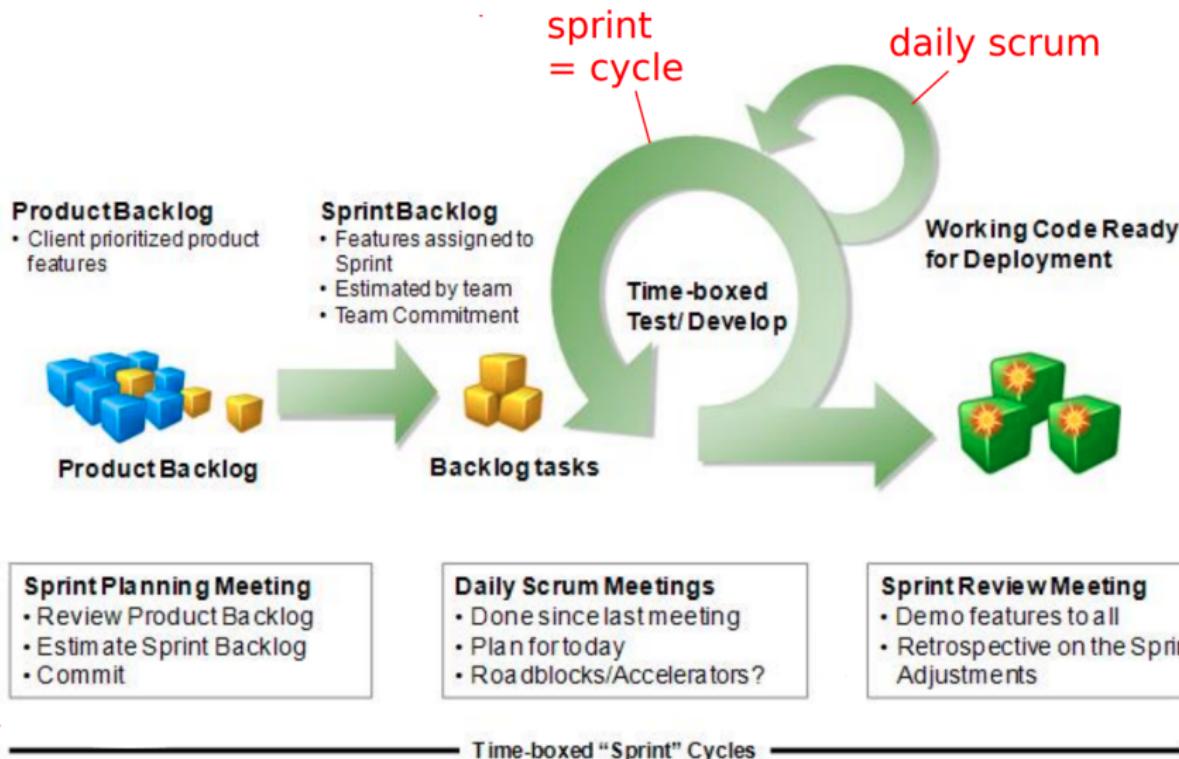
- ▶ Pillars come to life and build trust for everyone only when *commitment, courage, focus, openness* and *respect* are embodied and lived by the team.

Roles

- ▶ The **Product Owner** is responsible for maximising the value of the product and work of the Development Team.
- ▶ The **Development Team** is equipped with all necessary skills to successfully deliver the product.
- ▶ The **Scrum Master** supports for the Product Owner, the Development Team.

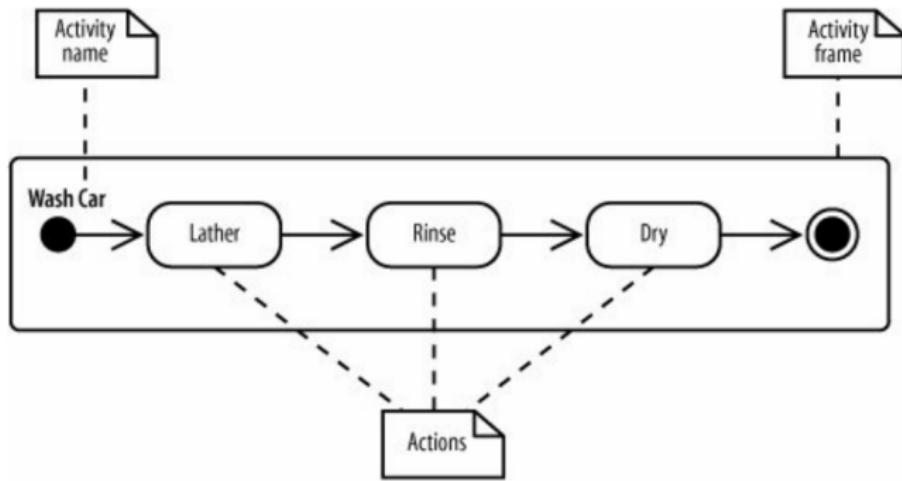
Events

- ▶ Sprint: time-box to produce a “done” product increment.
- ▶ Sprint Planning, Daily Scrum, Sprint Review, Sprint Retrospective.

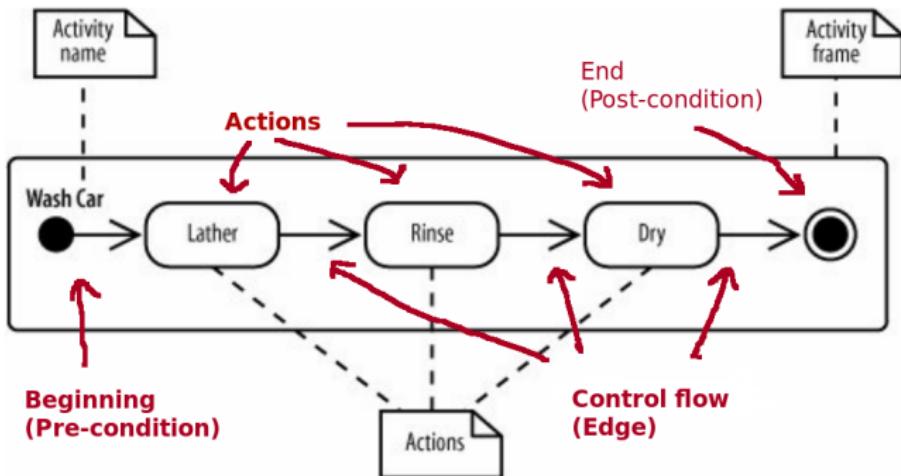


UML Activity Diagrams

Activity diagrams show *high-level actions* chained together to represent a **business process** occurring in our system.

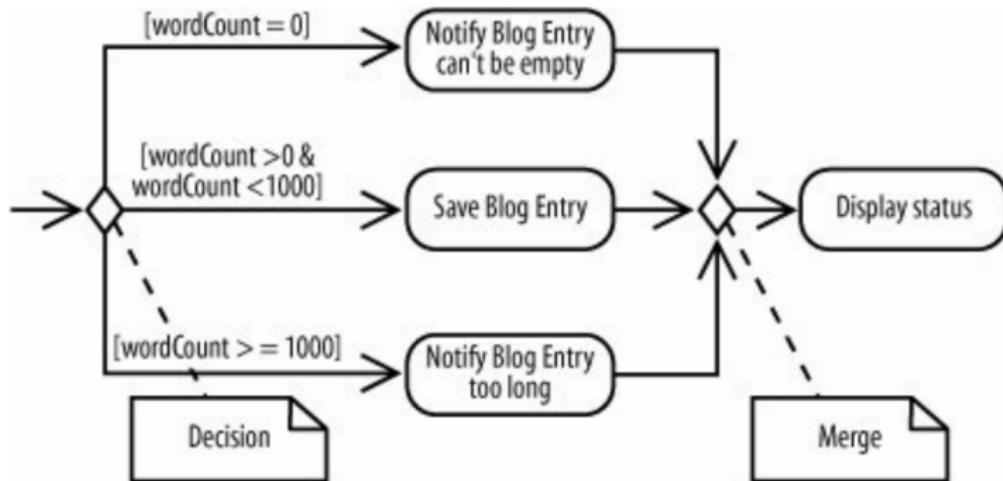


Activity diagrams show *high-level actions* chained together to represent a **business process** occurring in our system.



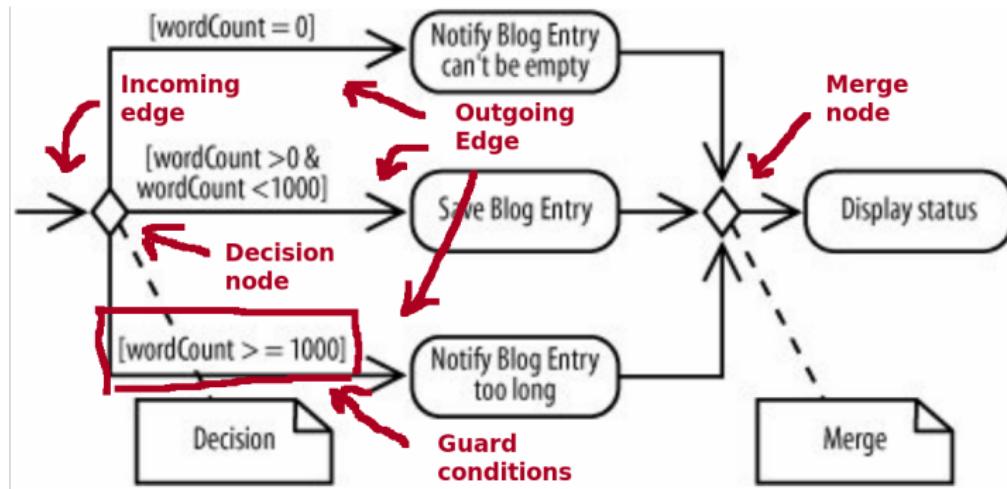
An action starts when it receives a **control token**.

Diamonds are used to depict **decision** or **merge nodes**.



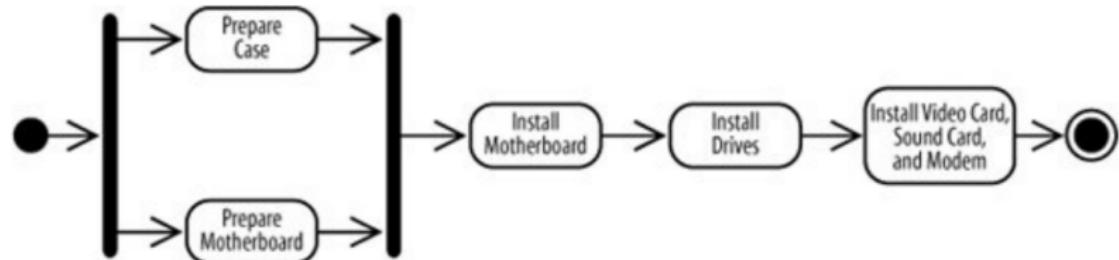
An action starts when it receives a **control token**.

Diamonds are used to depict **decision** or **merge nodes**.



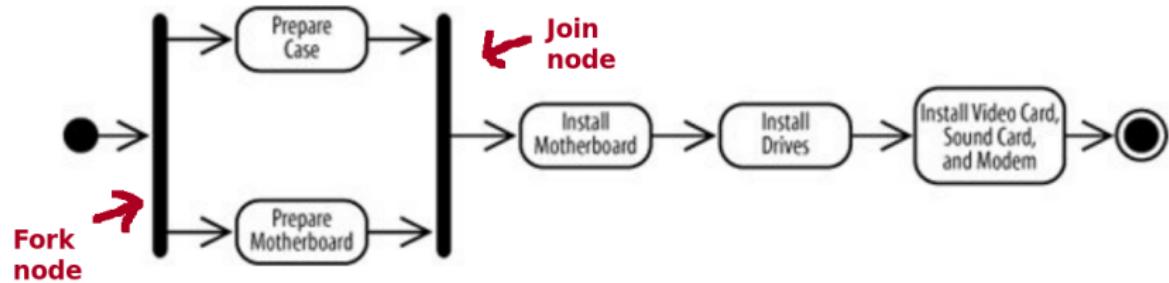
Multiple actions that run in parallel are depicted using **forks** and **joins**.

A flow is broken up into two or more *simultaneous flows*. All incoming actions must finish before the flow must proceed past the join.



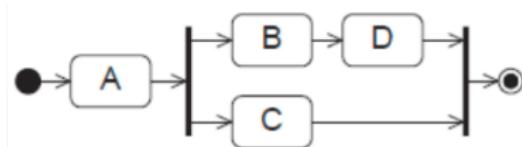
Multiple actions that run in parallel are depicted using **forks** and **joins**.

A flow is broken up into two or more *simultaneous flows*. All incoming actions must finish before the flow must proceed past the join.



A quiz question for you

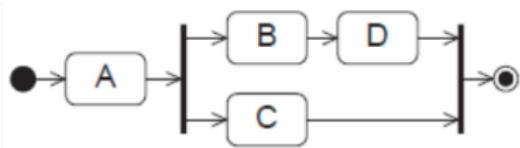
Consider the following UML activity diagram. Which of the following action sequences are possible during one execution of the activity diagram?



- a) A → B → D → C
- b) A → C → B → D
- c) A → B → C → D
- c) A → D → C → B

A quiz question for you

Consider the following UML activity diagram. Which of the following action sequences are possible during one execution of the activity diagram?



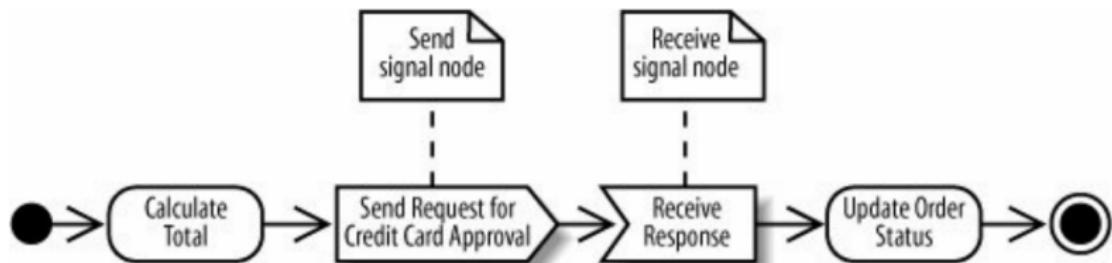
- a) **A → B → D → C**
- b) **A → C → B → D**
- c) **A → B → C → D**
- c) A → D → C → B

When time is a factor in an activity, **time events** are used to model a *waiting period*.



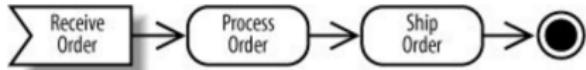
When an activity interacts with an *external actor or process*, messages are sent and received. These are called **signals**.

A **receive signal** wake up an action, whereas a **send signal** is a message to the external participant.



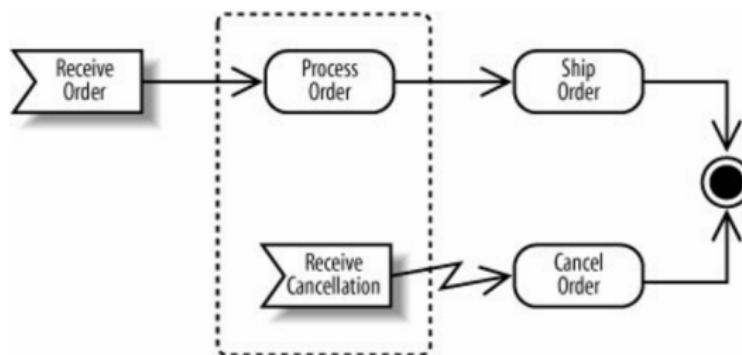
More on signals:

- ▶ When send and receive signals are combined, a **synchronous flow** is depicted.
- ▶ If only a send signal is depicted, the flow is **asynchronous** (does not wait for a response to proceed).
- ▶ A receive signal can replace a starting node.



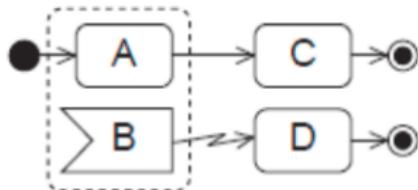
Interruptions are receive signals that cause activities to stop following their “expected” flows. A **lightning bolt** symbol is used to depict an interruption.

Interruptible regions show the area in which an interruption is expected to occur.



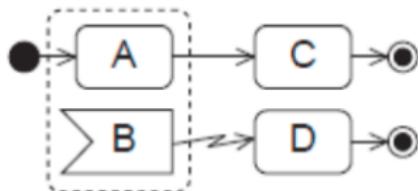
A quiz question for you

Consider the following UML activity diagram. Which of the following action sequences are possible during one execution of the activity diagram?



- a) A → C
- b) A → B → D
- c) A → C → B → D

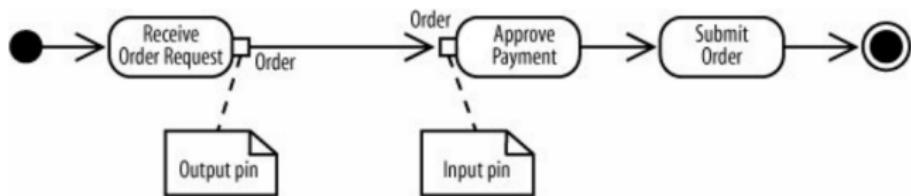
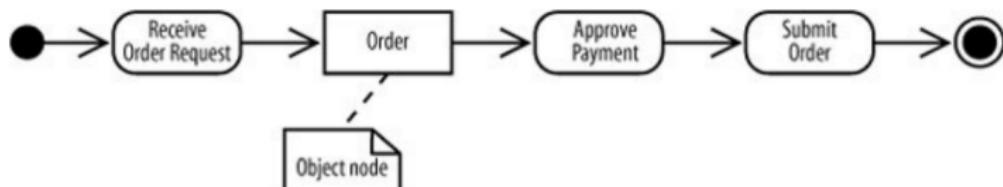
Consider the following UML activity diagram. Which of the following action sequences are possible during one execution of the activity diagram?



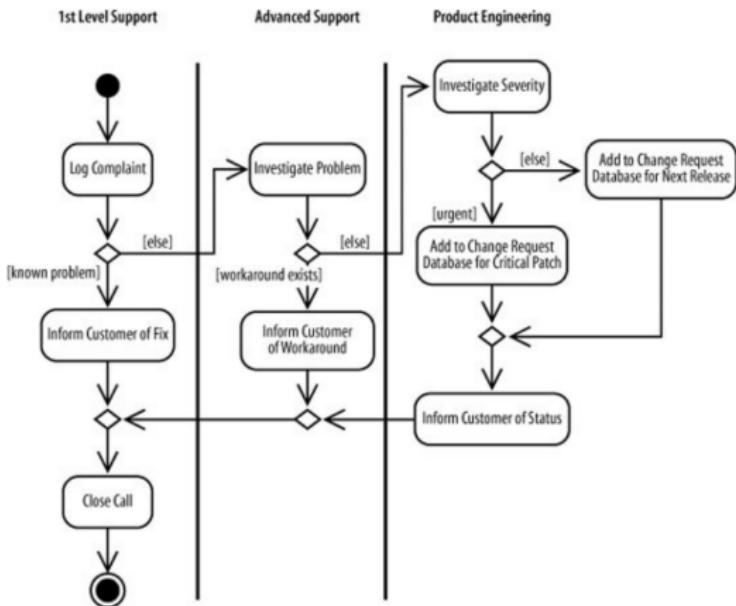
- a) **A → C**
- b) **A → B → D**
- c) A → C → B → D

Objects can be depicted either by using **object nodes** or **pins**.

Input pins represent input objects, i.e., an action cannot run without an object parameter. **Output pins** specify that an object is output from an action.



Swimlanes or partitions are used to depict which component or participant is responsible for which actions.



Today's learning outcomes:

- ▶ We discussed several Software Development Lifecycle processes.
- ▶ You were introduced to UML Activity Diagrams and their notation.

CS2SE: Software Engineering



Dr Alexandros Giagkos

School of Computer Science and Digital Technologies
College of Engineering and Physical Sciences

Subscribe to the board and **participate** in the discussions!

- ▶ Requirements Engineering
Feasibility study, Elicitation and Analysis of Requirements.
- ▶ Use Case Descriptions and UML Use Case diagrams.

- ▶ The requirements for a system are the descriptions of what the system should do, the services that it provides and the constraints on its operation.
- ▶ The process of finding out, analysing, documenting and checking these services and constraints is called **requirements engineering** (RE).
- ▶ The term *requirement* is used inconsistently in the literature.
 - ▶ **User requirements:** statements plus diagrams of system services and any associated constraints.
 - ▶ **System requirement:** detailed descriptions of system's functions, services and operational constraints.
 - ▶ **Domain** and **software** requirements (will be soon discussed).

► User Requirement Definition:

- 1.** *The LocalCare system shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month.*

► System Requirement Specification:

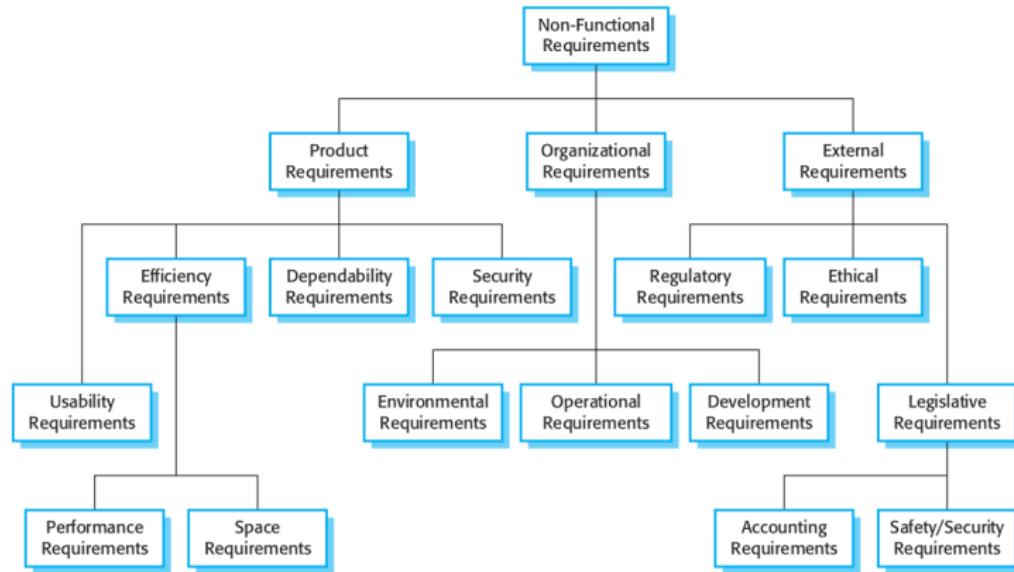
- 1.1** *On the last working day of each month, a summary of the drugs prescribed, their cost, and the prescribing clinics shall be generated.*
- 1.2** *The system shall automatically generate the report for printing after 17.30 on the last working day of the month.*
- 1.3** *A report shall be created for each clinic and shall list the individual drug names, the total number of prescriptions, the number of doses prescribed, and the total cost of the prescribed drugs.*
- 1.4** *If drugs are available in different dose units (e.g., 10 mg, 20 mg) separate reports shall be created for each dose unit.*
- 1.5** *Access to all cost reports shall be restricted to authorized users listed on a management access control list.*

Types of Requirements

- ▶ **Functional Requirements (FR):**
 - ▶ Statements of services a system should provide.
 - ▶ How the system should react to particular input.
 - ▶ How the system should behave in particular situations.
 - ▶ In some cases, a functional requirement may explicitly state what the system should NOT do.
- ▶ **Non-Functional Requirements (NFR):**
 - ▶ Constraints on services or functions offered by the system.
 - ▶ Timing constraints, development-related, imposed by standards, etc.
 - ▶ NF requirements often apply to the system as a whole.

- ▶ Describe what the system should do.
- ▶ Depend on the type of software and expected users.
- ▶ User requirements are described in an abstract way to be understood by system users.
- ▶ System requirements are more detailed (input/output, exceptions, etc.).
- ▶ FRs are written in a functional requirements specification document, which needs to be both *complete* and *consistent*.

- ▶ Often more critical than functional requirements.
- ▶ Types of non-functional requirements:



- ▶ Non-functional requirements must be *testable*, written quantitatively so that they can be objectively tested.
- ▶ Rewriting NFR:

The system should be easy to use by medical staff and should be organised in such a way that user errors are minimised.



Medical staff shall be able to use all the system functions after four hours of training. After this training, the average number of errors made by experienced users shall not exceed two per hour of system use.

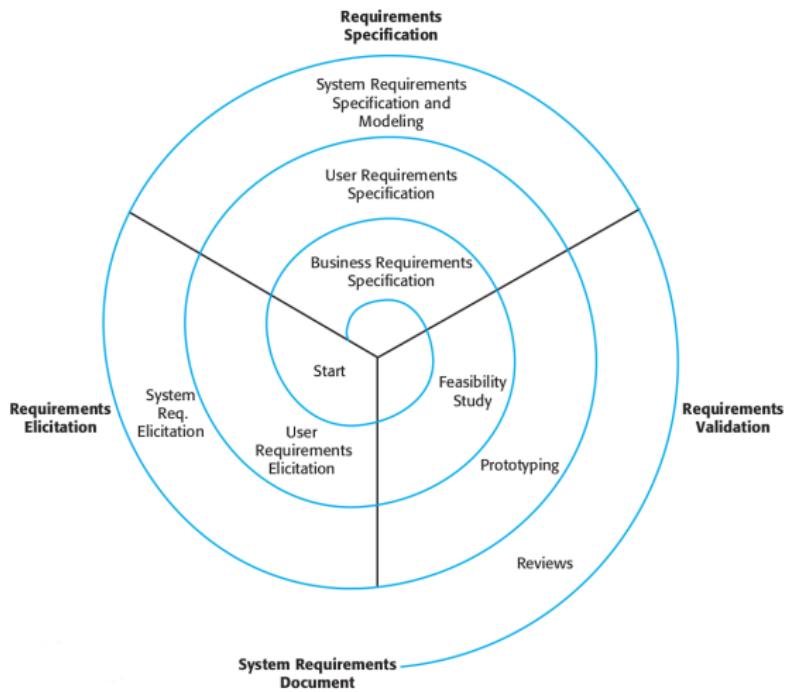
► Metrics for specifying NFR:

Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Mbytes Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

- ▶ Official statement of what software developers should implement. [[see example on Blackboard](#)]
- ▶ Covers both user and system requirements, FR and NFR.
- ▶ A thick document that has a diverse set of readers.

- ▶ Requirements specification is the process of writing down the requirements in the requirements document.
- ▶ Almost always written in natural language supplemented by diagrams and tables.
- ▶ Useful guidelines:
 - ▶ Invent a standard format.
 - ▶ Use language consistently to distinguish between mandatory and desirable requirements (e.g., “must/shall”, “should”, etc.).
 - ▶ Use text highlights (e.g., **bold**, *italic* or **colour**).
 - ▶ Avoid jargon, abbreviations and acronyms.
 - ▶ Explain rationale.

- ▶ A spiral view of the requirement engineering (iterative):



- ▶ **Feasibility study** is a short, focused study that assesses if the system is useful to the business.
- ▶ Should answer the following questions:
 - ▶ Does the system contribute to the overall objectives of the organisation?
 - ▶ Can the system be implemented within schedule and budget using current technology?
 - ▶ Can the system be integrated with other systems that are used?

Probe the feasibility of the following project:

Develop a system that allows real-time editing of a slide by all students at once when doing a lecture exercise.

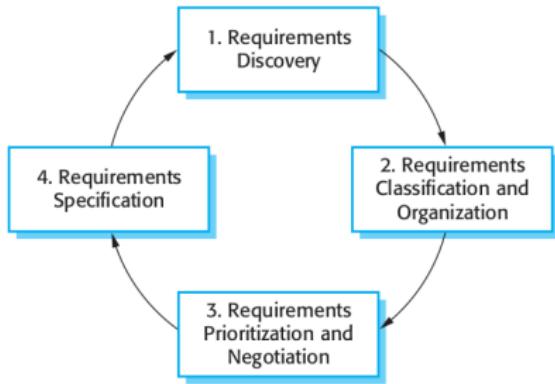
Probe the feasibility of the following project:

Develop a system that allows real-time editing of a slide by all students at once when doing a lecture exercise.

In general, one can use the following Checklist questions:

- ▶ What if the system wasn't implemented?
- ▶ What are current process problems?
- ▶ How will the proposed system help?
- ▶ What will be the integration problems?
- ▶ In new technology needed? What skills?
- ▶ What facilities must be supported by the proposed system?

- ▶ Software engineers work closely with stakeholders to elicit and analyse requirements.
- ▶ **Discovery**: applying several techniques to interact with stakeholders and find out about domain, user and system requirements.
- ▶ **Classification and Organisation**: grouping related requirements, making coherent clusters and associate them with sub-systems of the architecture.
- ▶ **Prioritisation and Negotiation**: ordering requirements and resolving conflicts.
- ▶ **Specification**: validating and formally documenting requirements.



- ▶ Stakeholders often do not know what they want/need.
- ▶ Requirements are often expressed in the stakeholders' own terms, and with implicit knowledge of their work.
- ▶ Different stakeholders have different requirements → conflicts.
- ▶ Political factors may influence the requirements of the system (e.g., managers may demand specific requirements to increase their own influence in the organisation).
- ▶ The importance of each requirement may change w.r.t. the economic and business environment.
- ▶ New stakeholders will come with a new set of requirements.

Several techniques, depending on the stakeholders:

- ▶ **Brainstorming**: creating new ideas – requires voting methods.
 - ▶ **Interviews**: closed or open, formal and informal.
 - ▶ **Questionnaires**: running short, targeted surveys.
 - ▶ **Examination of documents/artefacts**: reading current policies and procedures.
 - ▶ **Scenarios**: running example interaction sessions.
- Use cases**: textual or graphical using UML.
- ▶ **Prototyping**: best in receiving feedback, can be paper, powerpoint or functional.
 - ▶ **Ethnography**: watching daily activities and identifying problems that arise.

- ▶ Descriptions of example interaction sessions.
- ▶ Formally written but easy to grasp scenario that consists of:
 1. Assumptions on situation in which the scenario starts.
 2. Normal flow of events (primary path).
 3. What can go wrong and how to handle it (exceptions).
 4. Other activities that might be going on at the same time.
 5. Description of the system state when the scenario finishes.

- ▶ An example scenario – template may vary.

ATM withdrawal

Initial assumption: customer standing in front of ATM with a card

Normal: The customer inserts their card and the system swallows it and shows a prompt for PIN while checking with the bank whether the card is valid and what the account balance is. The customer inserts the correct PIN, waits for a menu of options and selects the “withdraw” option. The ATM then displays a selection of amounts. The customer selects one of these amounts and waits. The system displays a “wait” message, initiates a transaction from customer’s account to the ATM operation account. and after a short while ejects the card and a pile of cash. The customer collects both and the ATM briefly shows a thank you message.

What can go wrong: The card is stolen. The ATM reports this to the customer, retains the card, takes a picture of the customer and informs the police.

The ATM is out of banknotes. An “no cash available” warning shows in the beginning and the ATM does not offer the “withdraw” option. (+ many more)

Other activities: A thief may be eavesdropping to find out the customer’s PIN.

System state on completion: ATM displays a welcome screen for the next customer and is ready to accept their card.

- ▶ *Scenario* is a sequence of events and/or actions.
- ▶ *Workflow* is a set of scenarios with a common goal usually structured as follows:
 - ▶ Primary path: most common scenario.
 - ▶ Alternative path(s): less common scenarios to the goal.
 - ▶ Exception path(s): scenarios aiming for the goal but missing it.
- ▶ 80/20 rule applies: you'll spend 80% of your time dealing with what will happen 20% of the time (alternative paths).

The tea making process: teapot vs cup.

Primary path:

1. Fill kettle
2. Switch kettle on
3. Put tea in teapot
4. Pour boiling water
5. Wait for two minutes
6. Pour tea into cup
7. Add milk and sugar

Alternative path:

1. Fill kettle
2. Switch kettle on
3. Put teabag **in cup**
4. Pour boiling water
5. Wait for two minutes
6. Add milk and sugar

The tea making process: teapot using a faulty kettle.

Primary path:

1. Fill kettle
2. Switch kettle on
3. Put tea in teapot
4. Pour boiling water
5. Wait for two minutes
6. Pour tea into cup
7. Add milk and sugar

Exception path:

1. Fill kettle
2. Switch kettle on
3. **Kettle explodes**
4. Stop the fire

Note the numbering.

Primary path:

1. Fill kettle
2. Switch kettle on
3. Put tea in teapot
4. Pour boiling water
5. Wait for two minutes
6. Pour tea into cup
7. Add milk and sugar

Alternative path:

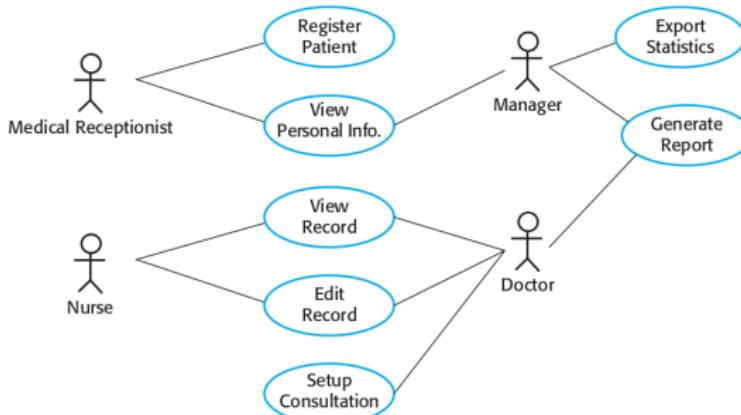
- 1.1 Kettle already full
 - Start with step 2
- 3.1 Cup instead of teapot
 - Put tea in cup at 3
 - Leave out step 6

Exception path:

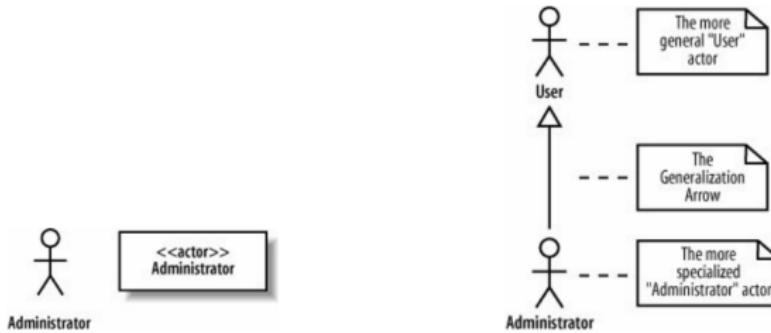
- 2.1 Kettle exploding
 - Kettle explodes after step 2
 - Leave out steps 3-7, stop the fire

Use Cases and Use Case Descriptions

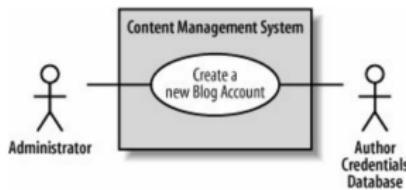
- ▶ A **use case** is a piece of functionality performed by the system that can be described by a short name.
- ▶ Each use case is associated with its **initiator(s)** and a **use case description**. [see template on Blackboard]
- ▶ Its name should describe the functionality from the initiator's point of view.



- Actors are drawn as *stick men* (human actors) or **stereotyped boxes** (external systems).

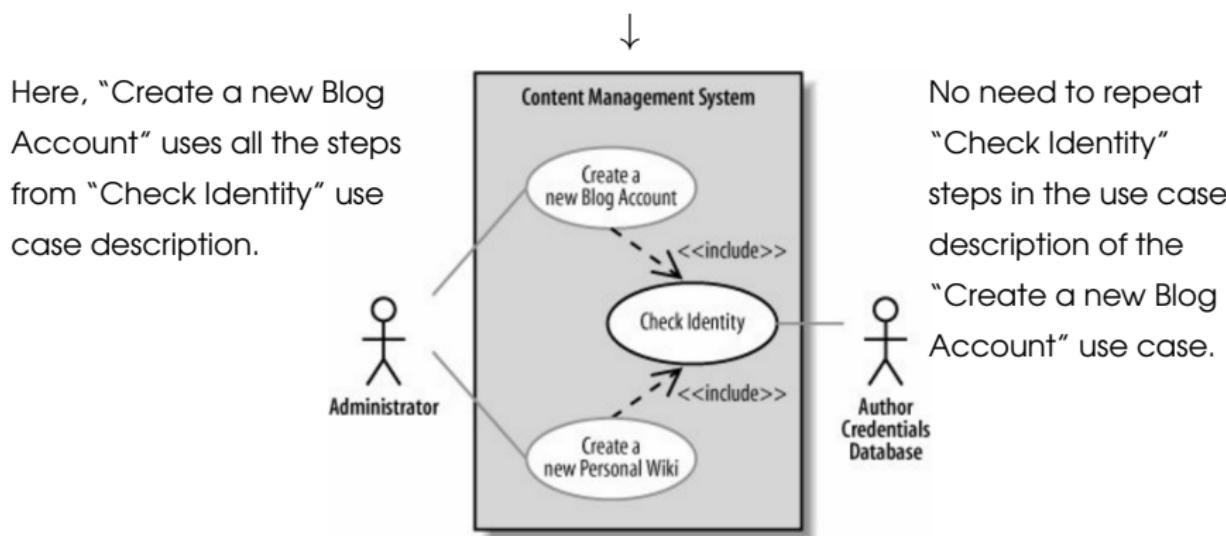


- A use case:

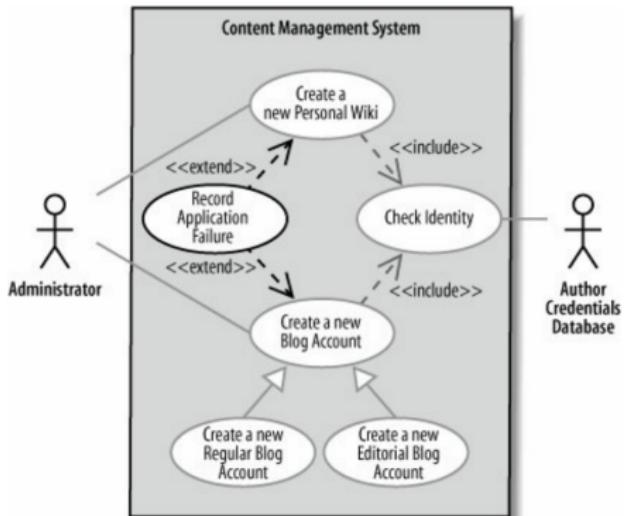


► The <<include>> relationship:

R1.1: *The content management system shall allow an administrator to create a new personal Wiki, provided the personal details of the applying author are verified using the Author Credentials Database.*



- ▶ <<Extend>> in this context means *optionally including*, i.e., may A may or may not include B.



"Create a new Blog Account" extends "Record Application Failure": the situation may sometimes happen.

Extend has nothing to do with Java's inheritance. For the latter, UML use case diagrams use the generalisation arrow.

- ▶ Group related requirements.
- ▶ Give them numbers so they are traceable, e.g., FR1, FR2, NFR1, etc.
- ▶ Decompose the system into sub-systems and components of related requirements.
- ▶ Define relationships between these components.
- ▶ Identify which design patterns to employ.

- ▶ Classifying by audience. Who are requirements written for?
- ▶ User requirements: stakeholders
- ▶ System requirements: stakeholders who engage in detail.
- ▶ Software requirements: mainly developers.

requirements type	client managers	system end users	client engineers	developer managers	system architects	system developers
user reqs	✓	✓	✓	✓	✓	
system reqs		✓?	✓		✓	✓
software specs			✓?		✓	✓

- ▶ **MoSCoW:** Must, Should, Could and Would (or Won't, Wish to) have.
- ▶ If any of the *must* requirements is not included, the delivery is considered a failure.
- ▶ *Should* requirements are important but not as time-critical as the *must* ones.
- ▶ Requirements related to improving user experience are often classified as *could*.
- ▶ *Won't* requirements are least-critical and can be delivered in the next release.

Assign requirements to the correct category.

Register

Book tickets

MUST

Buy refreshments

SHOULD

Refreshments reserved to seats

COULD

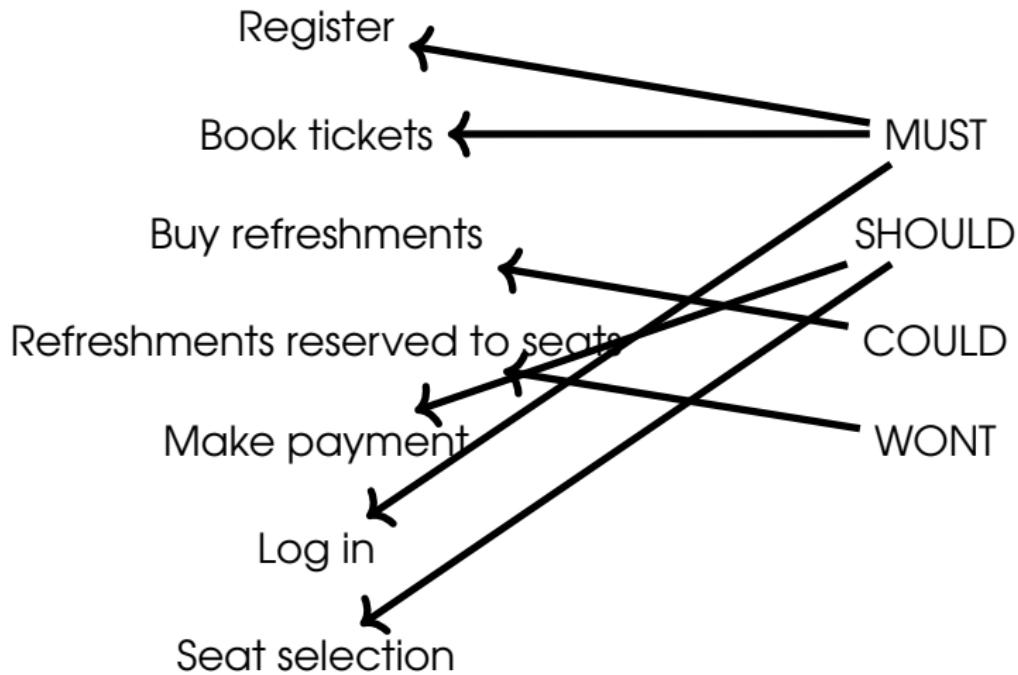
Make payment

WONT

Log in

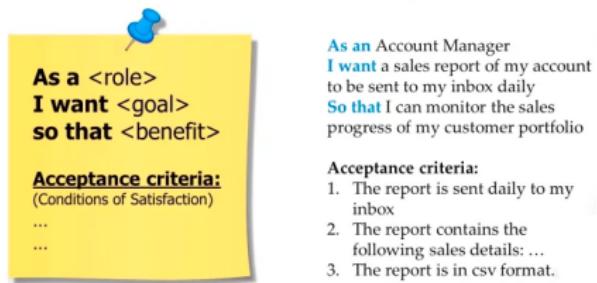
Seat selection

Assign requirements to the correct category.



- ▶ Rank requirements on an ordinal scale, using different numerical values based on importance.
- ▶ **Ranking** can be used with MoSCoW or any other similar method/variant.
- ▶ E.g., there are multiple *must* requirements, which one to implement first?

- ▶ XP's user stories are used instead of scenarios (use cases).

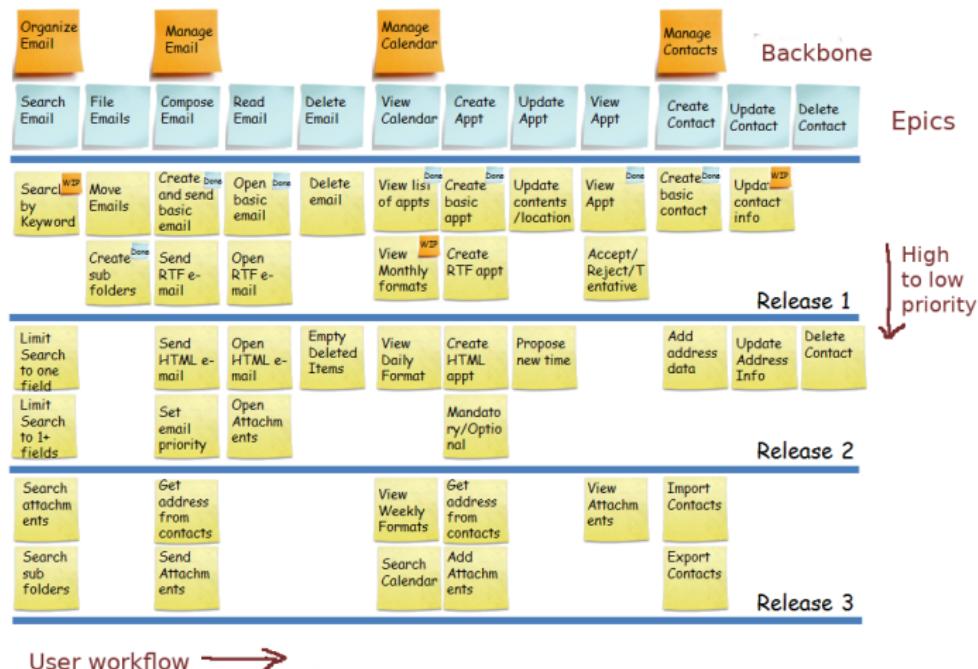


- ▶ Cards usually contain a checklist of tests to validate the requirement later.

- ▶ Good user stories are INVEST:
 - ▶ **Independent**: can be understood without having to read some other user story.
 - ▶ **Negotiable**: invitation to discuss, not a contract.
 - ▶ **Valuable**: giving useful outcomes to the stakeholders.
 - ▶ **Estimatable**: so we know they are small.
 - ▶ **Small**: fits in one cycle of development.
 - ▶ **Testable**: so it can be declared as “completed”.

Requirements Prioritisation with Agile (cont.)

- Prioritising user stories using user stories maps.



User story map taken from <https://www.infoq.com/presentations/user-story-map/>

- ▶ Requirements validation is the process of checking that requirements actually define the system that the customer really wants.
- ▶ Types of checks:
 - ▶ Validity check: does the system provide the functions which best support the needs of the stakeholders?
 - ▶ Consistency check: are there any conflicts to resolve?
 - ▶ Completeness: are all functions required by the customers included?
 - ▶ Realism: Can the requirements be implemented given time and budget?
 - ▶ Verifiability check: can the requirements be tested?

- ▶ Ways to conduct requirements validation:
 - ▶ **Requirements reviews:** the requirements are analysed systematically by a team of reviewers who check for errors and inconsistencies.
 - ▶ **Prototyping:** an executable model of the system in question is demonstrated to end-users and customers.
 - ▶ **Test-case generation:** if a test is difficult or impossible to design, the requirements will be difficult to implement.

Today's learning outcomes:

- ▶ We discussed the processes of Requirements Engineering.
 - ▶ Discovery, classification and organisation, prioritisation, specification.
- ▶ You were introduced to UML Use Case Diagrams.

CS2SE: Software Engineering



Dr Alexandros Giagkos

School of Computer Science and Digital Technologies
College of Engineering and Physical Sciences

Q: Where can we find past papers for this module?

There are no past papers; the module assessment changed from last year. We will have a 2:30hrs open-book computer-based examination in January (see mod specs for details).

It will include questions based on case studies that will be provided in advance. Lucy and I will give you more information in due course.

All questions will be based on lecture notes and tutorial exercises.

- ▶ Business/Domain/Context Analysis.
- ▶ UML Object, Class and State Machine Diagrams.

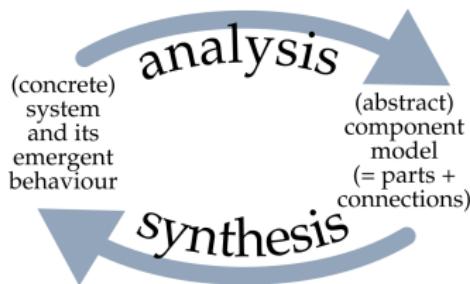
- ▶ Unit 1: **Systems Thinking** (systems, components and properties/behaviours), component diagrams.
- ▶ Unit 2: **SDLC** (Waterfall, UP, Agile), activity diagrams.
- ▶ Unit 3: **Requirements Engineering**, use case diagrams and descriptions.
- ▶ **Unit 4: Business Analysis, object/class/state machine diagrams.**
- ▶ *Unit 5: Systems Analysis, sequence/communication diagrams.*
- ▶ *Unit 6: Software Testing.*

Analysis:

- ▶ An investigation of the component parts of a whole.
- ▶ Involves some kind of *abstraction*, i.e., creating a component model.

Synthesis:

- ▶ Creating something real out of abstract models.



In Business/Context/Domain Analysis:

- ▶ Analyse existing structures and their processes.

In Requirements Analysis:

- ▶ Discover and define desirable external behaviour of a new or modified system.

In Systems Analysis:

- ▶ Elaborate requirements into a more formal specification of the desirable external behaviours (including user's perception of the internal behaviours).
- ▶ In reality, here we do more synthesis than analysis!

Think about creating an activity diagram during a system's design. Would you call it an analysis or synthesis activity?

- ▶ Analysis.
- ▶ Synthesis.

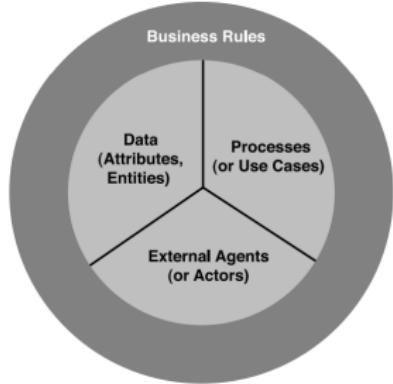
Think about creating an activity diagram during a system's design. Would you call it an analysis or synthesis activity?

- ▶ Analysis.
- ▶ Synthesis.

The activity diagram describes a process that (hopefully!) fulfils a desired requirement. When we draw activity diagrams during system design, we do not model an existing process of the organisation.

- ▶ Business Analysis is the set of tasks and techniques we use when working with stakeholders to understand the structure, policies and operations of an organisation.
- ▶ A Business Analyst (BA):
 - ▶ Identifies the business problems and opportunities.
 - ▶ Elicits the needs and constraints from stakeholders.
 - ▶ Analyses the needs and defines requirements.
 - ▶ Assesses and validates the potential and actual solutions.
- ▶ Main focus on business goals and not technology designs.
- ▶ BAs to Developers ratio 1/6, but now changing towards BAs.

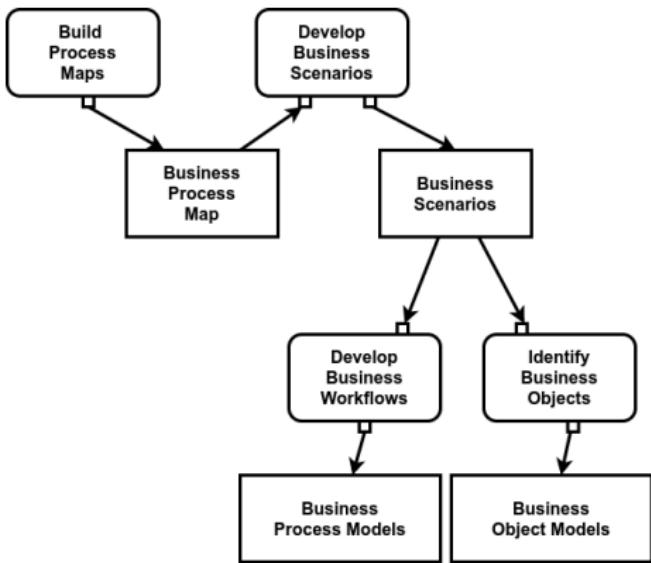
- ▶ When describing a business, there are four basic requirements components.
- ▶ Information is *data*.
- ▶ Manual or automated activities and procedures the business performs.
- ▶ People (or systems, or departments) inside and outside the organisation.
- ▶ Guidelines, constraints and policies under which the organisation operates.

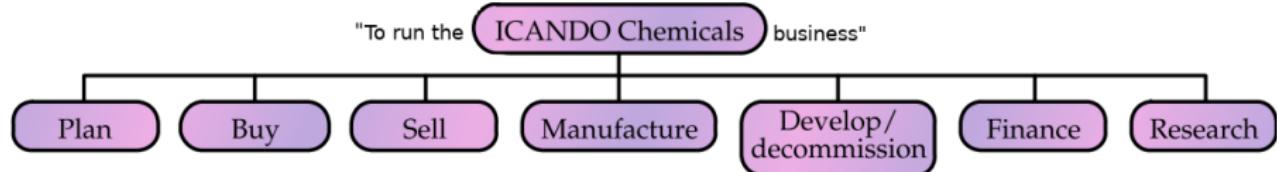


- ▶ The BA breaks requirements down and “sees” the specific parts of the business that may need improvement.
 - ▶ More specific questions lead to more detailed requirements.
- ▶ Why document requirements?
 - ▶ People forget!
 - ▶ Verbal communication is fraught with errors.
 - ▶ People answer the same question differently if asked twice.
 - ▶ Writing down something forces us to think of it carefully.
 - ▶ Stakeholders review what BAs write down.
 - ▶ New people joining a project get up-to-date sooner.

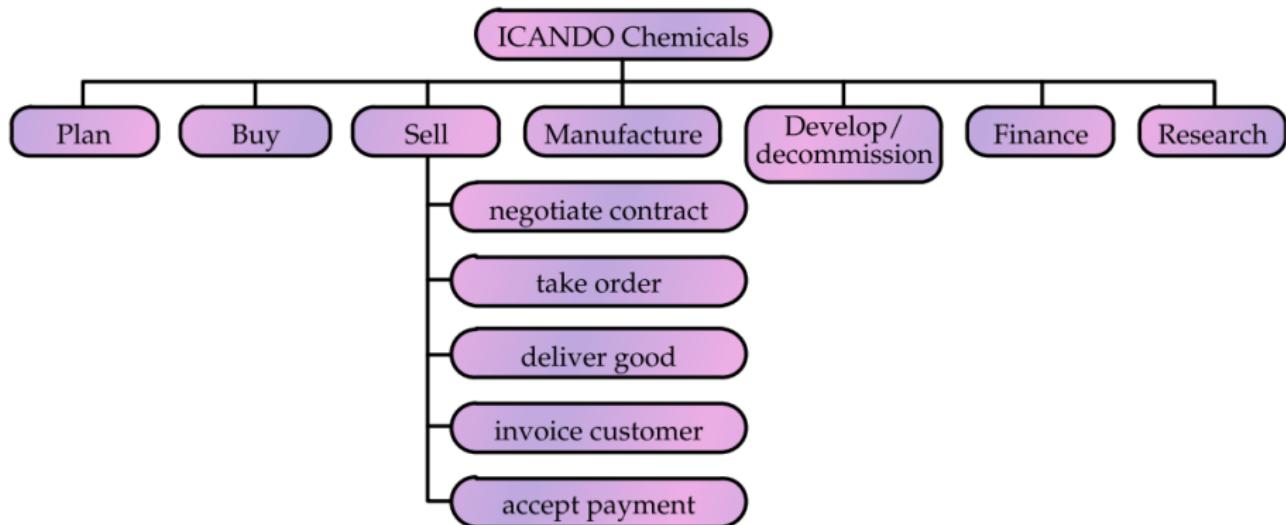
Conducting Business Analysis

- ▶ We have learned about business **scenarios**.
- ▶ We have seen **workflows** (primary/alternative paths in use case descriptions).
- ▶ We are familiar with **process models** (activity diagrams).
- ▶ We will look at:
 - ▶ **Business Process Maps.**
 - ▶ **Object Models** (class & object diagrams and state machines).



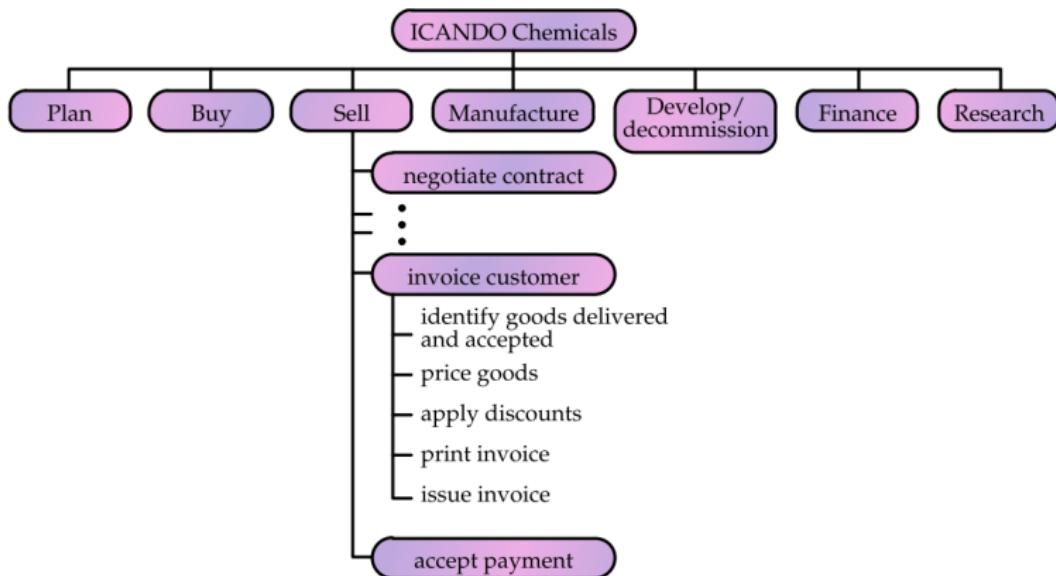


- ▶ A **business process** is a logical grouping of events that can be agreed as a fundamental element of the business.
- ▶ A **business process map** is a collection of named processes grouped hierarchically on no more than 3 levels.
- ▶ A few rules of thumb:
 - ▶ The root is the most abstract process.
 - ▶ Level 1 should not have more than 10 high-level processes.



- ▶ A lower-level process is a specialisation of its parent.

Business Process Maps (cont.)



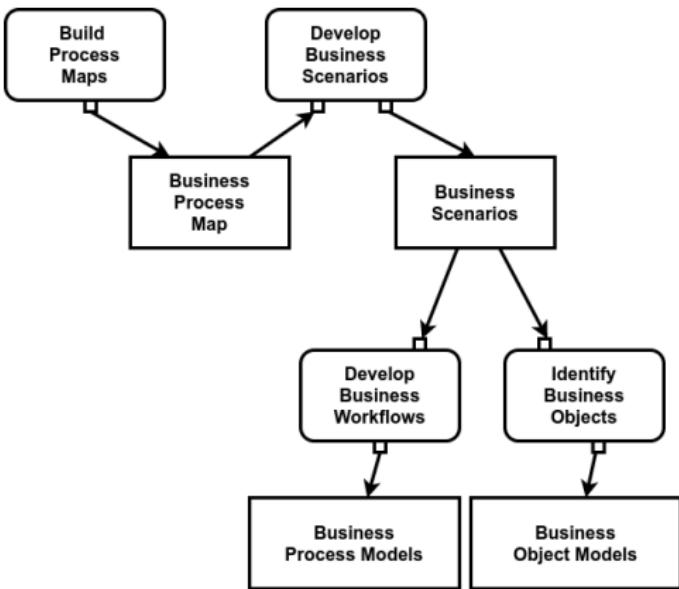
- Breaking down to more than 3 levels will render the business process map useless (it has to remain abstract!).

- ▶ Business process maps help us focus on main functionality.
 - ▶ Provide a top-level view and give scope, clarify terminology, easy to understand for everybody.
- ▶ No need to present a perfect break down.
- ▶ A business process map does not show:
 - ▶ The structure of the organisation.
 - ▶ Information flow.
 - ▶ Time considerations.
 - ▶ Interactions among processes.

- ▶ Usually a half-a-day workshop attended by business managers and experts.
 - ▶ All contribute to knowledge, but need to reach an agreement.
- ▶ Need for a *facilitator*:
 - ▶ Oversees the discussions and forces rules.
 - ▶ Aiming for a high-level view, and a good but not perfect map (not always achievable).
 - ▶ Ensures timing (time-boxed event).

Business Process Maps ✓

How to produce **Business Object Models** is next.



- ▶ Noun-verb analysis: the first step to class decomposition.
 - ▶ To break a large problem down into a class structure.
- ▶ Nouns may represent:
 - ▶ Classes.
 - ▶ Specialisations (classes that extend Abstract classes).
 - ▶ Attributes (i.e., a Student has a name).
 - ▶ Data (values to attribute variables/containers/etc.).
- ▶ Verbs may represent:
 - ▶ Services (i.e., methods) provided by a Class.
 - ▶ Services used by Classes.

Considering the workflows on the right, identify potential classes using the noun-verb analysis method.

Answer

Note the numbering.

Primary path:

1. Fill kettle
2. Switch kettle on
3. Put tea in teapot
4. Pour boiling water
5. Wait for two minutes
6. Pour tea into cup
7. Add milk and sugar

Alternative path:

- 1.1 Kettle already full
Start with step 2
- 3.1 Cup instead of teapot
Put tea in cup at 3
Leave out step 6

Exception path:

- 3.2 Kettle exploding
Kettle explodes after step 2
Leave out steps 3-7, stop the fire

Considering the workflows on the right, identify potential classes using the noun-verb analysis method.

Answer

- ▶ kettle
- ▶ teabag
- ▶ teapot
- ▶ cup
- ▶ water (amount of water)
- ▶ sugar (-//-)
- ▶ milk (-//-)
- ▶ minute (meta-level)
- ▶ step (-//-)

Note the numbering.

Primary path:

1. Fill kettle
2. Switch kettle on
3. Put tea in teapot
4. Pour boiling water
5. Wait for two minutes
6. Pour tea into cup
7. Add milk and sugar

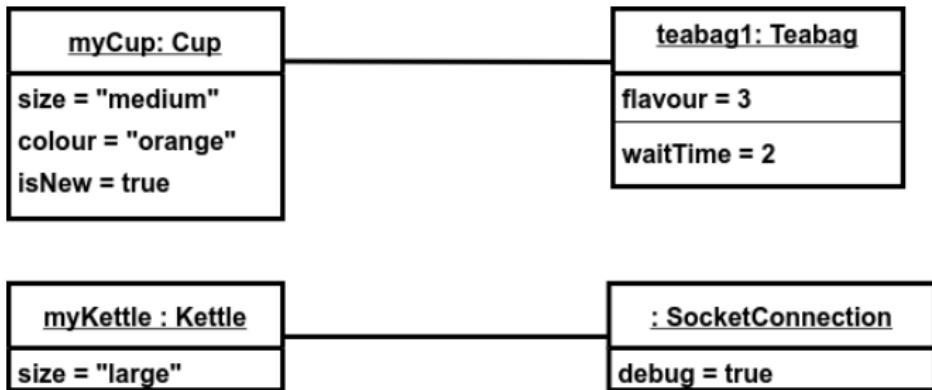
Alternative path:

- 1.1 Kettle already full
Start with step 2
- 3.1 Cup instead of teapot
Put tea in cup at 3
Leave out step 6

Exception path:

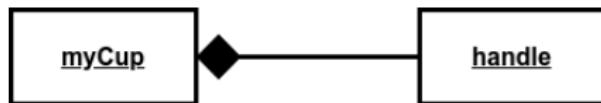
- 3.2 Kettle exploding
Kettle explodes after step 2
Leave out steps 3-7, stop the fire

- ▶ UML Object Diagrams are used to communicate object information (relationships and static view).
 - ▶ They represent instances of classes and derive from [UML Class Diagrams](#).
 - ▶ Object diagrams may or may not contain object attribute values.
 - ▶ Class diagrams are usually more detailed, include attribute and method information.
- ▶ Notation:  myCup is an instance of class Cup.

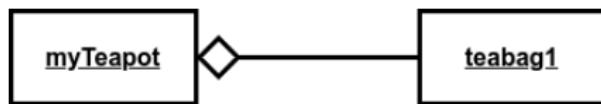


- ▶ Anonymous objects may exist, can you think why?

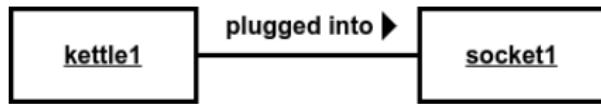
- ▶ **Composition** (part does not exist without the whole):



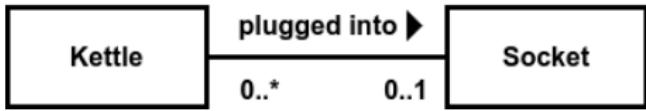
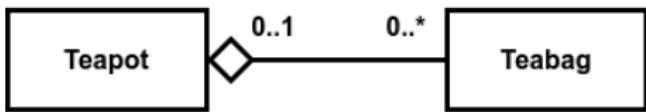
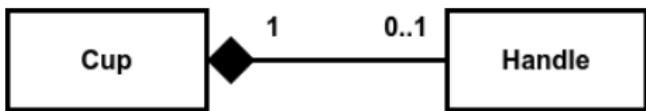
- ▶ **Aggregation** (a part of a whole):



- ▶ **Association** (arbitrary relations, can be navigable):



- ▶ Multiplicities should be added to provide more information about relationships when showing classes of a system.



(a) Draw an object diagram showing relations among: a shirt, one of its buttons and one of its seams.

Show a *composition*, an *aggregation* and an *association* other than aggregation.

(b) Draw a class diagram showing the same relations between shirts, buttons and seams in general.

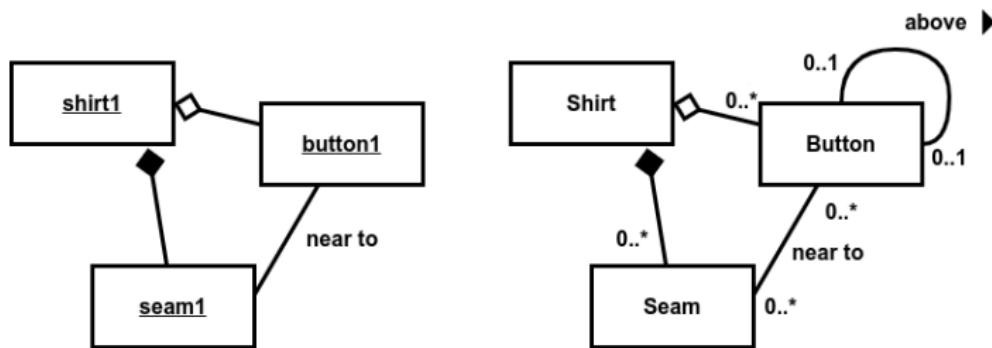
Include *multiplicities* where appropriate.

(a) Draw an object diagram showing relations among: a shirt, one of its buttons and one of its seams.

Show a *composition*, an *aggregation* and an *association* other than aggregation.

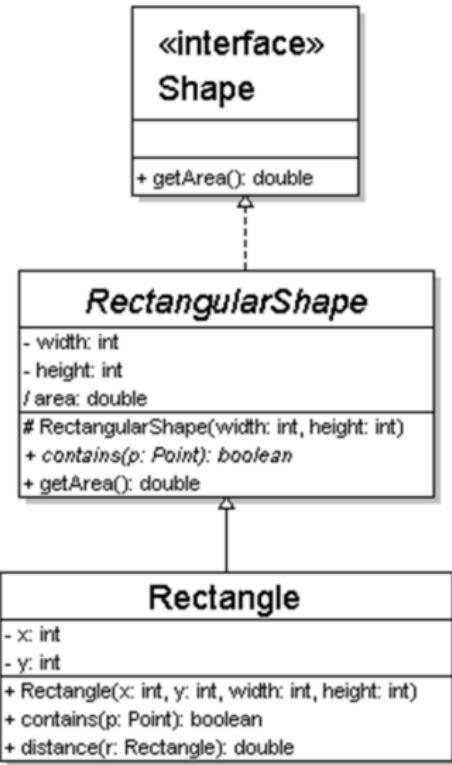
(b) Draw a class diagram showing the same relations between shirts, buttons and seams in general.

Include *multiplicities* where appropriate.



A Detailed Class Diagram

- ▶ Class diagrams can be very detailed.
- ▶ - private, + public,
protected, / derived.
- ▶ Parameters' and return types.
- ▶ Interface and Abstract classes (realisation and specialisation arrows).



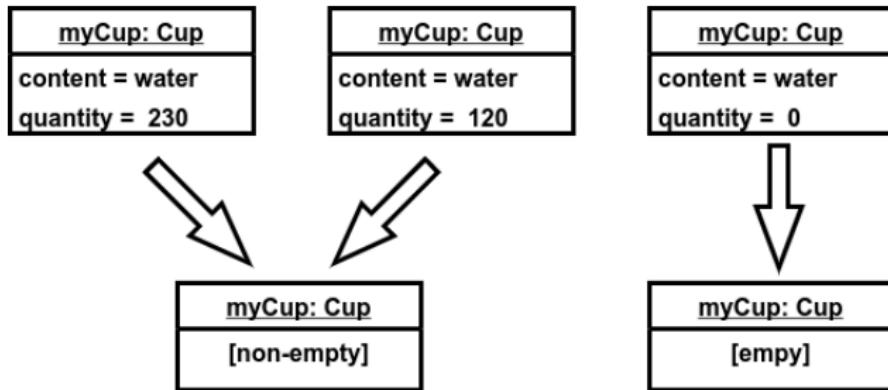
Use a UML Object Diagram to:

- ▶ Show a static view of an interaction.
- ▶ Describe object relationships of a system.

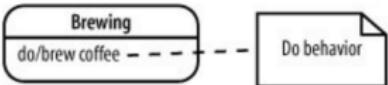
Use a UML Class Diagram to:

- ▶ Better understand the general overview of the schematics of an application.
- ▶ Provide an implementation-independent description of types used in a system.

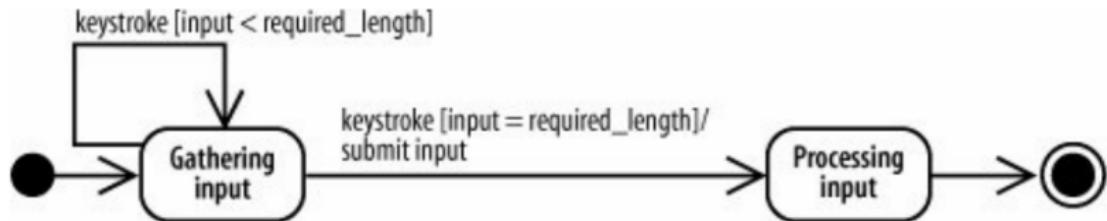
- ▶ State machine diagrams are used to show how an object's *state* changes during its lifetime.
- ▶ *Object state* = attribute values.
- ▶ Drawing a state machine requires increase of abstraction.



- Object states can be a passive quality (on or off) or an active quality, i.e., the object is *doing* something.

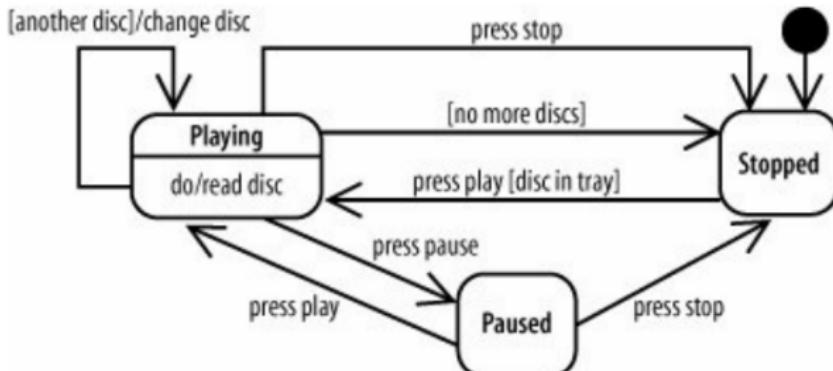


- Objects *transition* from *source* states to *target* states.

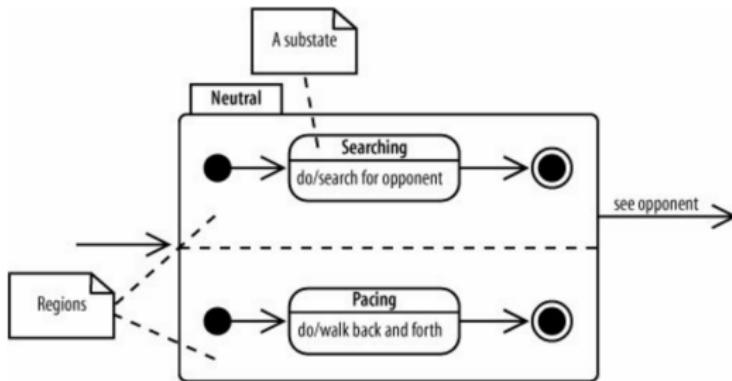


- A transition syntax: *trigger* *[guard]* / *behaviour*.

- More than one reasons for a transition.

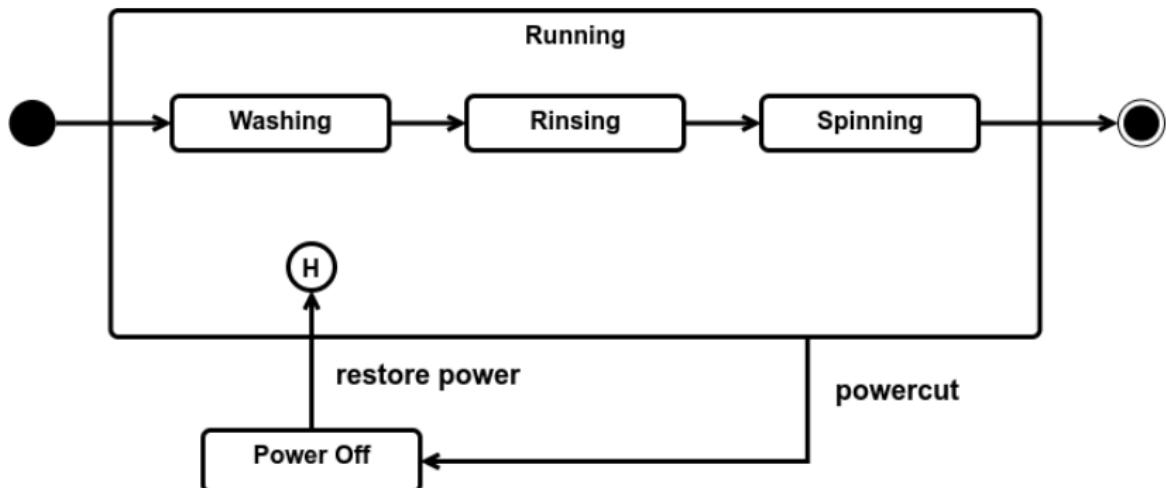


- ▶ UML allows concurrent states to be shown using the **composite states** notation.
- ▶ A non-player game character is in the “Neutral” state, which consists of two substates “searching” and “pacing”.

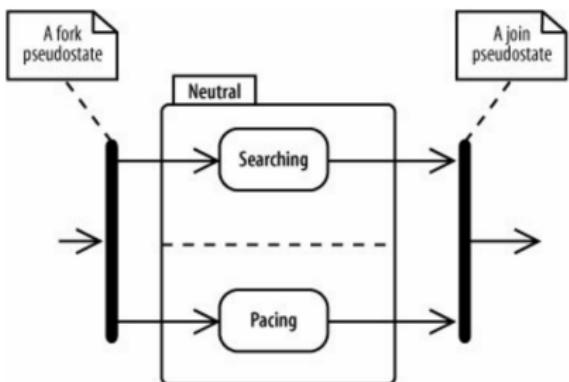
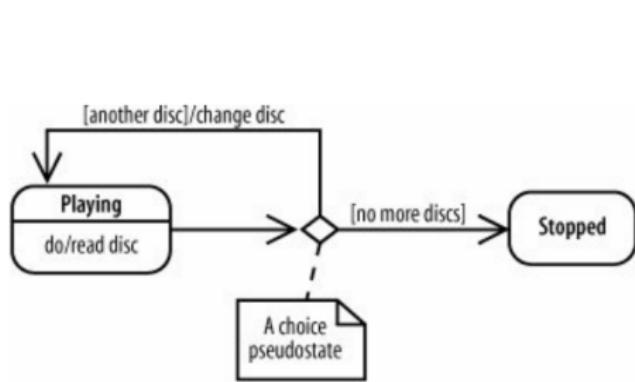


- ▶ Even if a substate runs to completion, the composite state is complete when every substate diagram is complete.

- ▶ History states are used to remember the state before an interruption happened.



- UML notations allows *choices*, *forks* and *joins* to be possible.



Use a UML state machine to:

- ▶ Show event-driven objects in a reactive system.
- ▶ Describe how an object moves through various states within its lifetime.

Today's learning outcomes:

- ▶ We discussed Business Analysis.
 - ▶ Discovery, classification and organisation, prioritisation, specification.
- ▶ You introduced to UML Object, Class and State Machine Diagrams.

CS2SE: Software Engineering



Dr Alexandros Giagkos

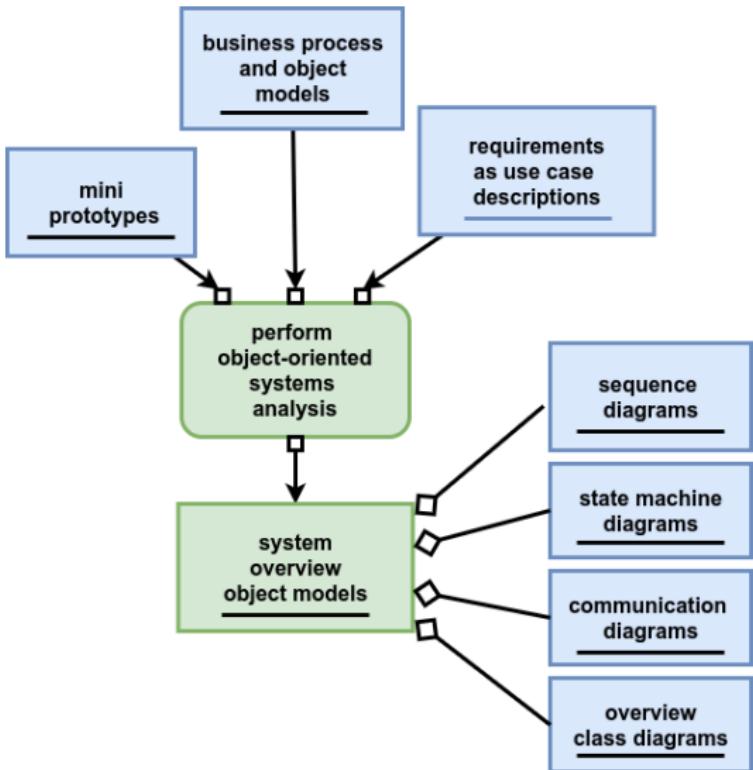
School of Computer Science and Digital Technologies
College of Engineering and Physical Sciences

- ▶ Systems Analysis.
- ▶ UML Sequence and Communication Diagrams.
- ▶ Practical work based on a case study.

- ▶ Systems Analysis is being conducted when we draft technical solutions to our requirements.
- ▶ The deliverable is not yet implementable, but very close to be.
- ▶ Overlaps with Domain Analysis, but is more technical and definitely a synthesis activity.

The big picture

- ▶ Domain Analysis feeds Systems Analysis with information.
- ▶ Systems Analysis delivers a more technical solution to meet the requirements.
- ▶ i.e., detailed models of the system, modelling of internal mechanisms, etc.

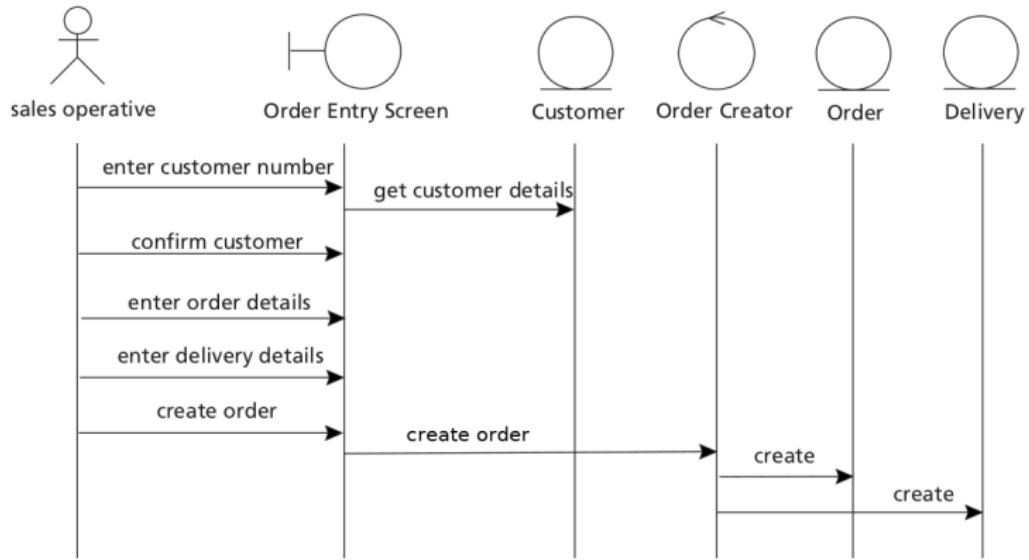


- ▶ The *object-oriented Systems Analyst* (SA) role is to:
 - ▶ Research problems and determine requirements.
 - ▶ Plan technical solutions for an enhanced or new system.
 - ▶ **Derive an outline object design from the requirements.**
 - ▶ **The design realises all use cases via actor-object and object-object collaboration.**
 - ▶ **Objects of the design have a specific responsibility that can be implemented.**
 - ▶ From business processes and their use cases to a comprehensive model of the system's objects, behaviours and interfaces.

- ▶ Software-based object-oriented modelling was first introduced to simulate real-world entities (Simula in the 60s).
- ▶ *Real-world entities*: software objects representing the entities (e.g., employees, students, orders, goods).
- ▶ *Attributes of entities*: object fields (e.g., employee name).
- ▶ *Behaviours*:
 - ▶ *Autonomous behaviours*: object's own thread executes object's private methods.
 - ▶ *Object interactions*: signals sent between objects.
- ▶ The essence of OO models: **objects** interact by passing **messages** to each other.

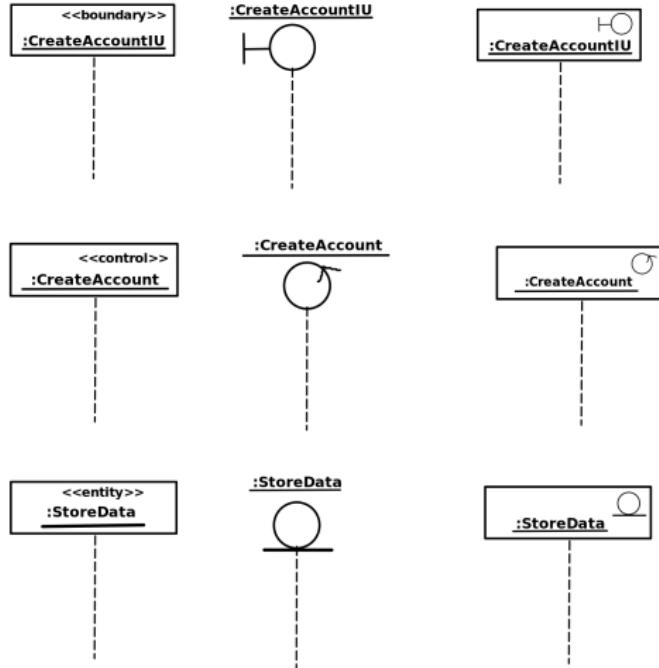
- ▶ UML Sequence Diagrams provide the necessary notation to draw objects and sequences of interactions between them.
- ▶ It is a two-dimensional diagram: objects are arranged along the top of the diagram, and the interactions are listed underneath in time order going down the page.
- ▶ Using analysis, they help us tie together the sequences in use case descriptions.
- ▶ Notation for three types of objects available.

An example of UML Sequence Diagram



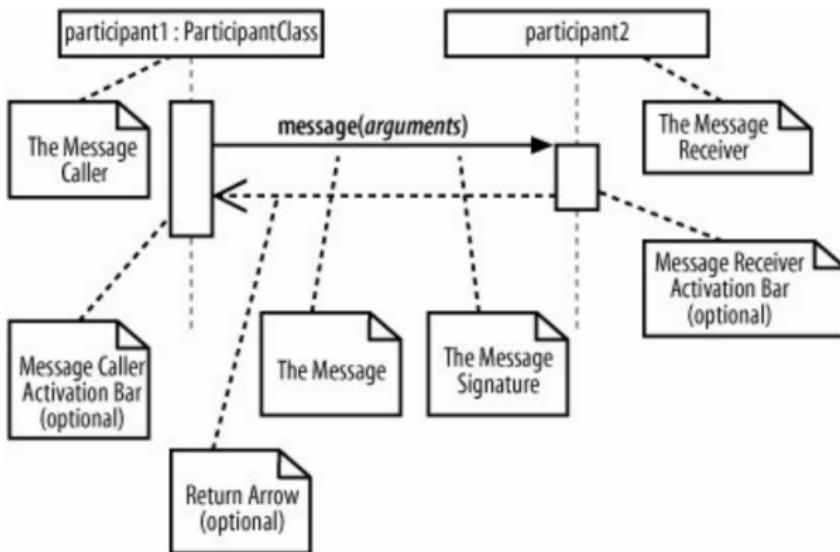
- ▶ Boundary Objects
 - ▶ Facilitate interactions with the outside world (usually an UI, and API of some sort).
 - ▶ Manage the dialogue between an actor and the system.
 - ▶ Do not store any data, but may wrap it appropriately.
 - ▶ At least one is always present in a sequence diagram.
- ▶ Control Object
 - ▶ Realise use cases and organise complex behaviours.
 - ▶ Interact with boundary objects to send/receive input/output and interact with actors.
- ▶ Entity Objects
 - ▶ Model a store or persistence mechanism that captures information (data) in the system.

Notation of Objects in Sequence Diagrams



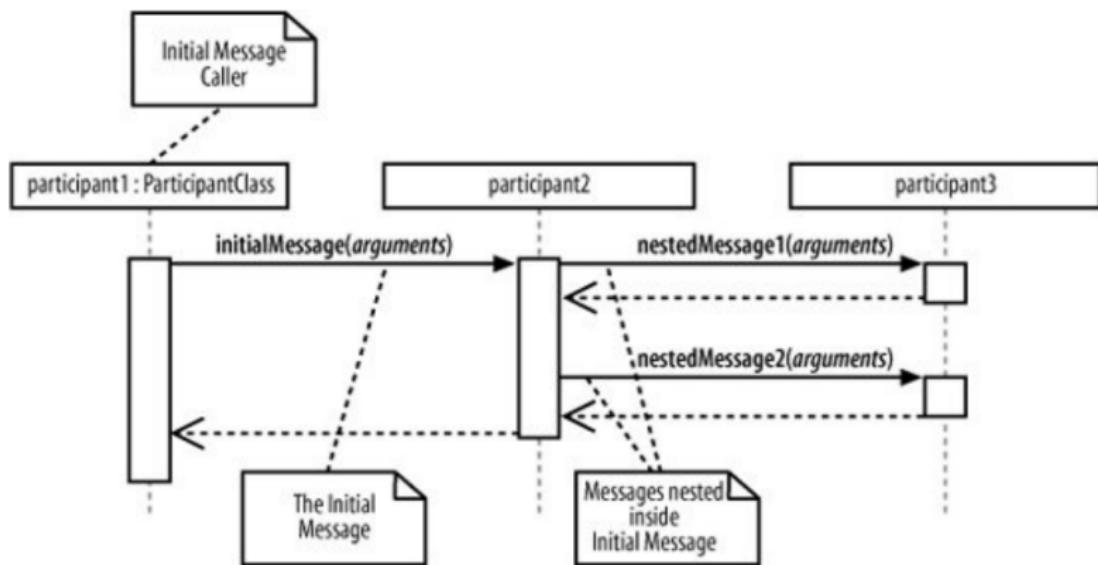
*As always in UML,
decide which
notation is better for
you and be consistent.*

UML Sequence Diagram: Activation Bars



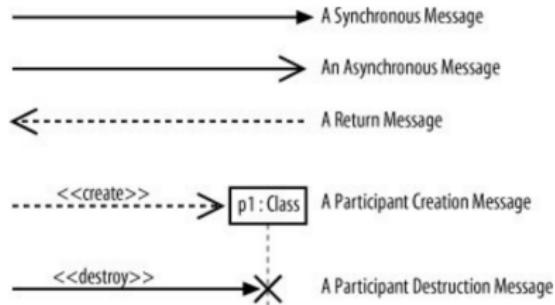
- ▶ *Activation bars* indicate that the object is active (i.e., is doing something).
- ▶ Can be omitted as they can clutter up the diagram.

UML Sequence Diagram: Messages

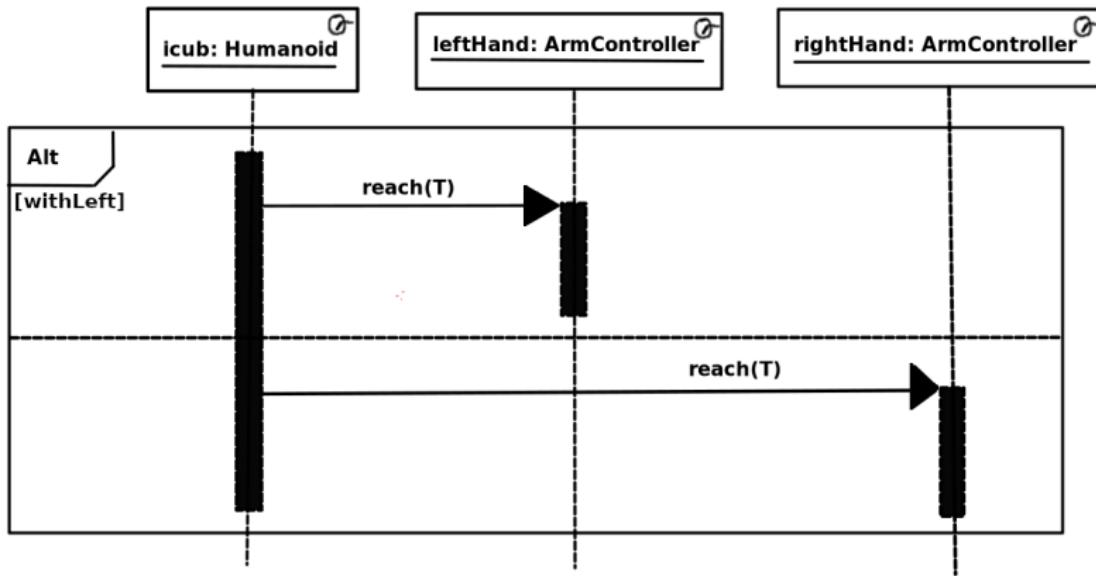


- ▶ Nested messages are allowed.

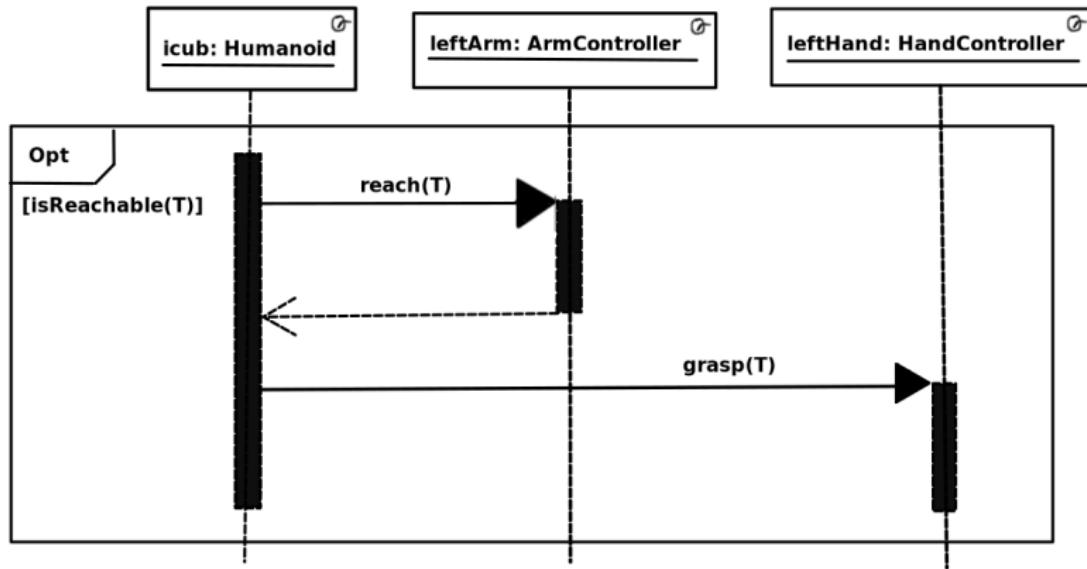
- ▶ Different types of messages can be shown.
- ▶ **Synchronous messages**: caller waits until the receiver finishes and returns.
- ▶ **Asynchronous messages**: “fire and forget”.



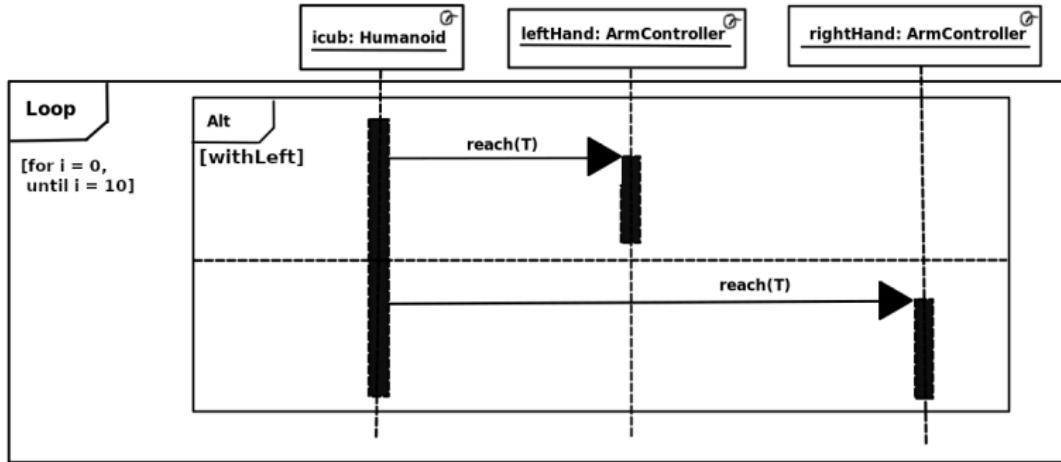
- Alternative fragments show a choice of behaviour in a workflow. A guard is used to evaluate a choice. Only one of the options is executed.



- Optional fragments are only executed when their condition is true.

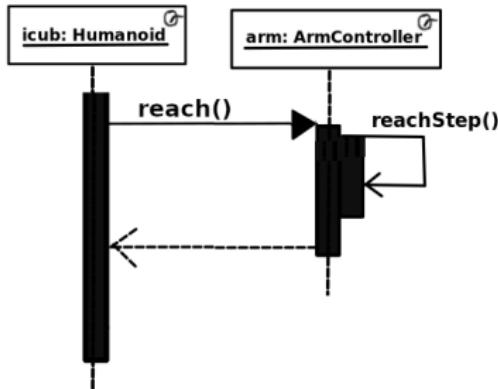


- ▶ Loop fragments illustrate repetitive sequences.



- ▶ Fragments can be combined to communicate the appropriate models.

- **Reflexive** calls allow an object to send messages to itself.



Process:

- ▶ For each use case:
 - 1 Pick a few representative scenarios, i.e., primary, alternative and exception paths.
 - 2 Express each scenario/path as a sequence diagram, identifying participating domain and system objects and their interactions.
 - 3 Rework the sequence diagrams as communication diagrams.
 - 4 Combine all communication diagrams to derive a outline/overview class diagram.

This is an example process to get a preliminary OO design, you will probably end up using other more streamlined and more iterative processes.

► Consider the following use case's primary path:

1. The sales operative takes the customer number and enters it on the screen.
2. The customer details are retrieved and displayed on the screen.
3. The sales operative checks that the customer details match those given by the customer, and ticks a confirm box.
4. The sales operative enters the order details.
5. The sales operative enters the delivery details.
6. The sales operative requests that the order is created.

► Finding actors and objects:

1. The sales operative takes the customer number and enters it on the screen.
2. The customer details are retrieved and displayed on the screen.
3. The sales operative checks that the customer details match those given by the customer, and ticks a confirm box.
4. The sales operative enters the order details.
5. The sales operative enters the delivery details.
6. The sales operative requests that the order is created.

Actors: SalesOperative

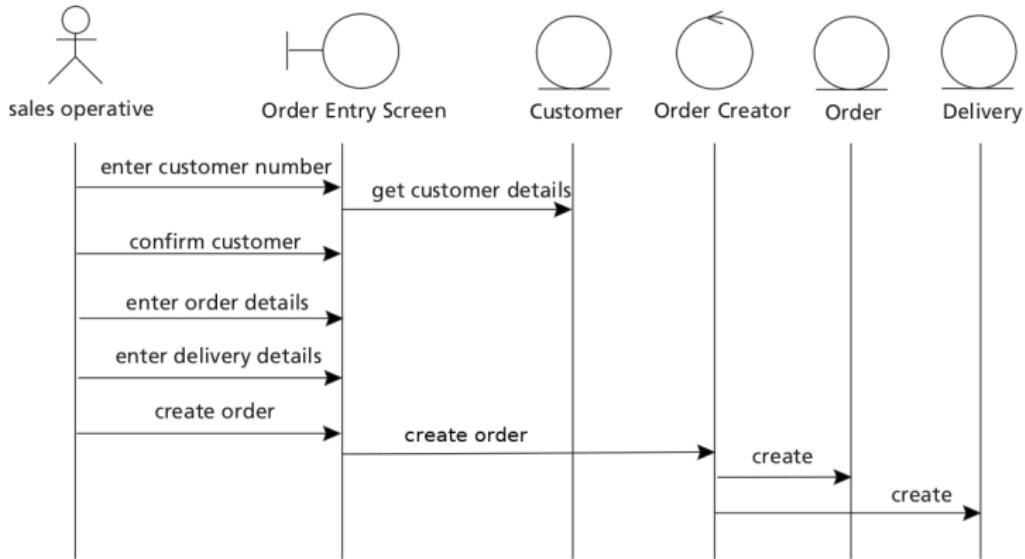
Boundary objects: Screen (to display details)

Entity objects: Customer, Order, Delivery

Process:

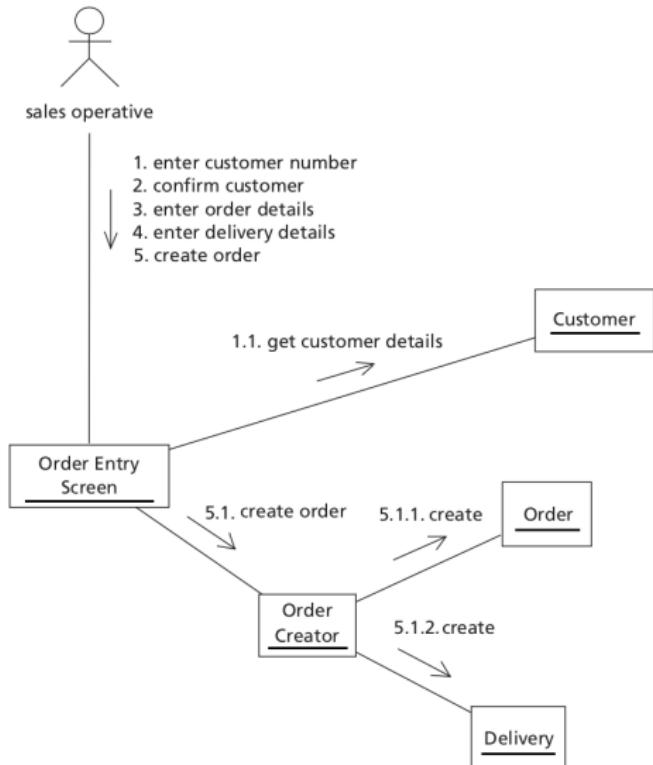
- ▶ Draw a lifeline on the left of the initiating actor.
- ▶ Draw next to it lifeline(s) for the actor's boundary object(s).
- ▶ Draw (generously spaced) interactions between actor and the boundary object(s).
- ▶ For each boundary object interaction determine whether there is a need to involve another object.
 - ▶ If the interaction is about a single entity object, create it.
 - ▶ If the system needs to perform an action that involves more than one entity object, consider creating a control object to manage the action.

Resulting Sequence Diagram

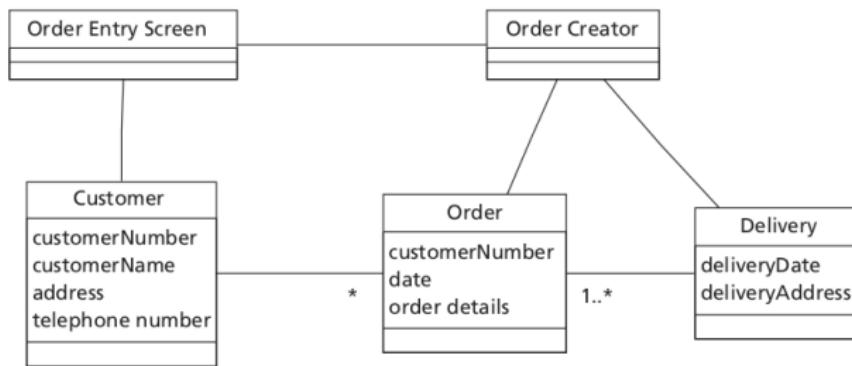


- ▶ A **Control Object** **OrderCreator** is responsible for the creation of an **Order** and the preparation of the **Delivery**.

- ▶ UML communication diagrams are another way of interrelating objects in sequence diagrams.
- ▶ Numbering interactions in sequence diagrams translates them easier to communication diagrams.
- ▶ Easier to derive class from communication diagrams.



- ▶ Knowing the objects, one can derive a UML class diagram and add further details such as attributes, associations and multiplicities.
- ▶ Note that some of these details may need clarification with the business analyst/stakeholders.



- ▶ The process of taking your design from use case and use case descriptions, to sequence diagrams, to communication diagrams and ultimately to class diagrams is iterative.
- ▶ At the end, we try to aggregate all the generated models into one outline class model of the software system.
- ▶ Nothing stops you from using other UML diagrams to generate your models. E.g., state machine diagrams are often used in Systems Analysis to model Actor — GUI interactions.

Today's learning outcomes:

- ▶ We discussed Systems Analysis.
- ▶ You were introduced to UML Sequence and Collaboration/Communication Diagrams.
- ▶ We discussed the process of deriving a systems analysis outline class diagram from use cases.

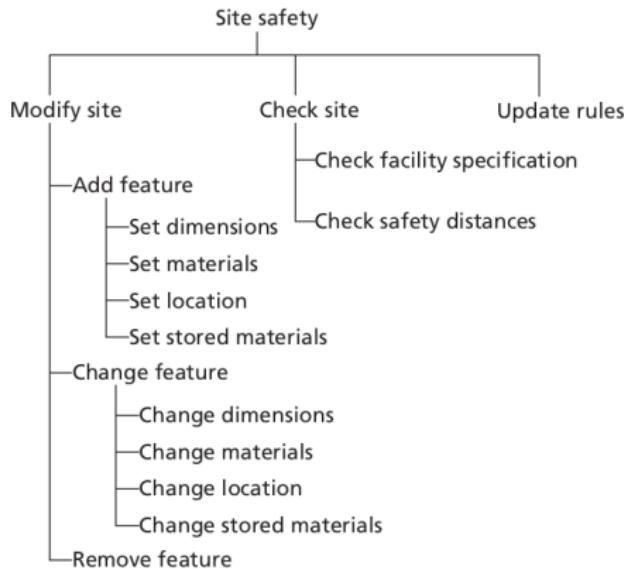
- ▶ Tutorial 5 is about deriving a class diagram from a use case related to the ICANDO Chemicals business we looked at last week.
- ▶ ICANDO Chemicals has a number of sites to produce and store chemicals.
- ▶ They want to ensure safety, thus their sites need to conform to rigorous safety standards.
- ▶ They want to develop a drawing tool that allows engineers to quickly draw a site, including all the storage facilities, offices, boundaries and roads.

- ▶ The software system must:
 - ▶ Store site maps, support their creation and editing.
 - ▶ Store safety rules, support their loading.
 - ▶ Check whether site maps conform to rules.
 - ▶ Visually report any rule breaches found.
 - ▶ Produce audit reports as proofs that sites comply with rules.

- ▶ Some work has been done already. The IT has conducted domain analysis and feasibility study.

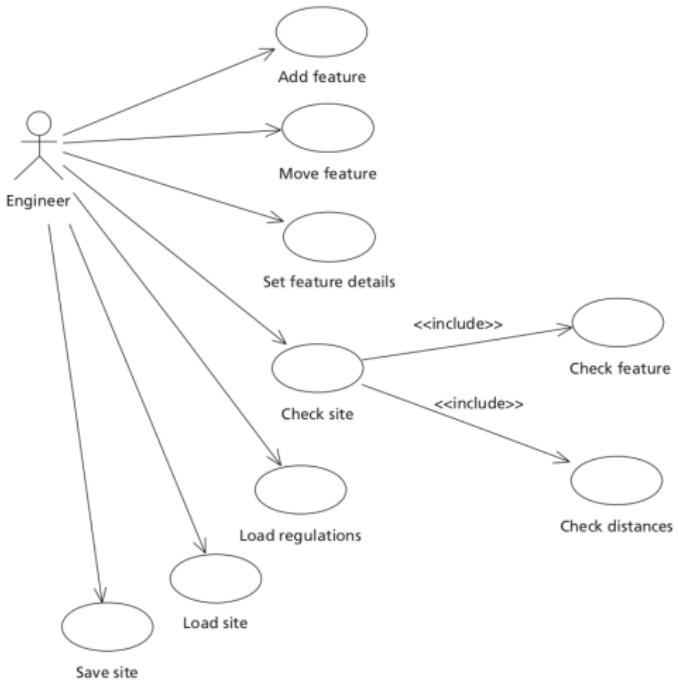
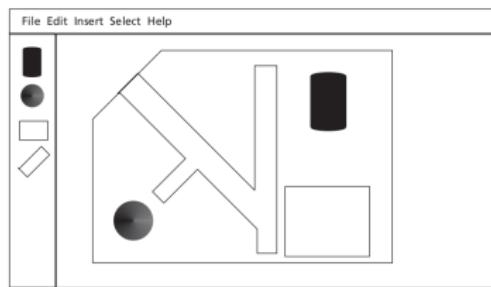
Stakeholder category	Role in project	Project implications	Actions needed
Research board	Fund and monitor project	Ensure value of project, and monitor results	Proposal and project review through normal research procedures
Central services audit team	Proposers of project	Will be keen to monitor project and adopt any successful result	Need to be engaged in analysis and design Need to be involved in evaluation
	Providers of expertise	Must be engaged in development	Programme of interviews and acquisition of technical documentation
Site engineers	Final users of system	Will determine ultimate acceptance	Need to be engaged in analysis, design and evaluation

- ▶ They drafted business process map:



Preparation for Tutorial 5 (cont.)

- They also drafted a prototype of the interface (left) and use case diagram (right).



CS2SE: Software Engineering



Dr Alexandros Giagkos

School of Computer Science and Digital Technologies
College of Engineering and Physical Sciences

- ▶ Software Testing.
 - ▶ Importance.
 - ▶ Black-box vs white-box testing.
 - ▶ Testing techniques.
 - ▶ Levels of testing.
 - ▶ Testing documentation and planning.
 - ▶ JUnit examples.
 - ▶ Test-Driven Development.
- ▶ Tutorials: Bring your laptops as there will be code (JUnit5).

- ▶ A process to **validate** a software application.
- ▶ *Observing, recording* and *comparing* the behaviour (output) of the software with the *intended* behaviour and output.
- ▶ Main purpose of testing is to try to *find errors*.
 - ▶ Is it always done?

- ▶ Testing should be done **early** and **often**.
- ▶ No need to wait until all operations of a software component are fully implemented.
- ▶ Even a smallest change on a piece of code may *invalidate* the entire system!
 - ▶ Testing should be done **whenever changes have been made**.

Test Data

- ▶ Test data should test the software at its limits and test *business rules*.
- ▶ Test data should include:
 - ▶ extreme values
(e.g., out-of-bound);
 - ▶ borderline values
(e.g., -1, 0.999);
 - ▶ invalid combinations of values
(e.g., age=3, marital status=married);
 - ▶ nonsensical values
(e.g., negative order line quantities);
 - ▶ heavy loads
(i.e., are performance requirements met?)

- ▶ Ideally, test scripts should be built and executed by *specialist test teams* who have access to software.
- ▶ Testing should also be done by business and systems analysts.
- ▶ In eXtreme Programming (XP), programmers are expected to write **test harnesses** for classes before they write the code.
- ▶ Intended users of the system should test against requirements, aka **user acceptance testing**.

What kind of testing?

- ▶ **Black box testing** seeks to establish whether the end-product:
 - ▶ Does what it's meant to do.
 - ▶ Performs tasks as fast as it should.
 - ▶ Checking FR, and is usually done by Systems Analysts.
-
- ▶ **White box testing** seeks to establish whether the end-product:
 - ▶ Delivers a good solution, not just a solution to the problem.
 - ▶ Checking NFR, and is usually done by the developers.

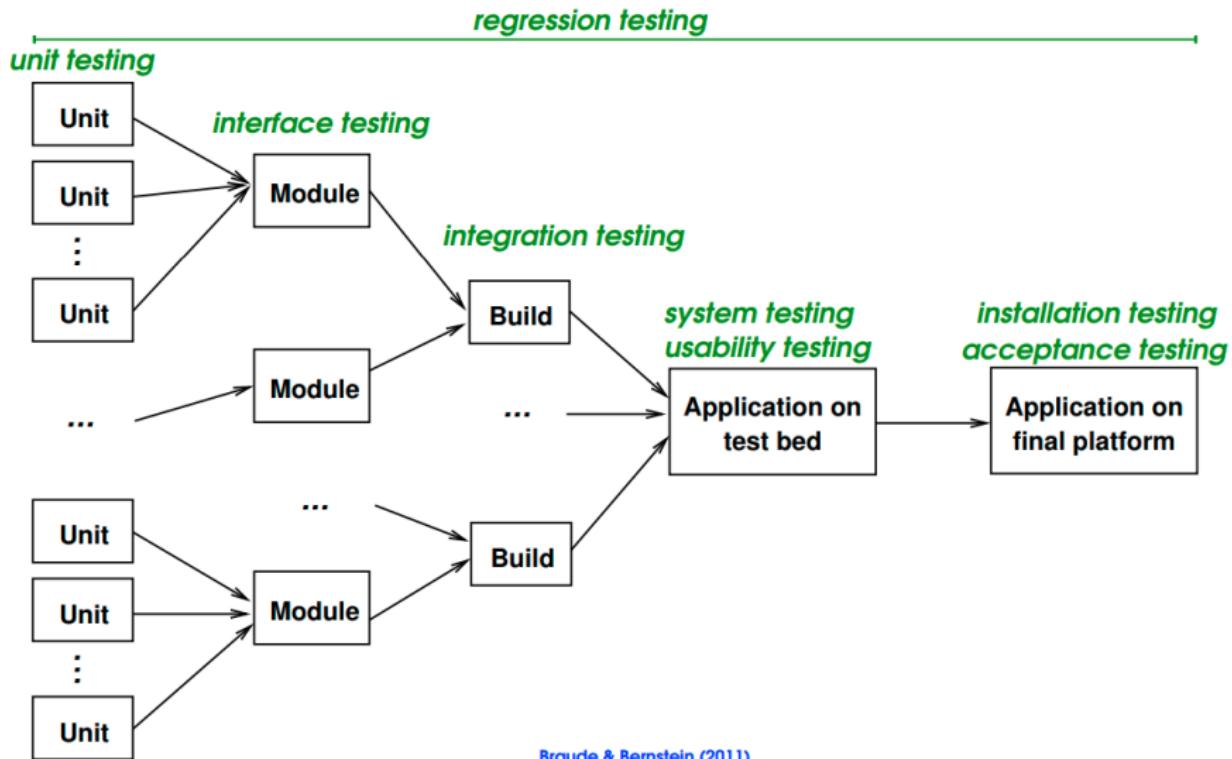
- ▶ **Unit testing:** ensures that individual classes work correctly.
- ▶ **Interface testing:** validates the functions exposed by modules.
- ▶ **Integration testing:** establishes whether all classes in the system work correctly together.
- ▶ **Subsystem testing:** checks if each subsystem works correctly and delivers the required functionality.
- ▶ **System testing:** checks if the *WHOLE* system works seamlessly.

- ▶ **Acceptance testing:** checks if the system works as required by the users and according to specification.
- ▶ **Usability testing:** tests if the intended users are satisfied with the software application in addressing its intended purpose.
- ▶ **Regression testing:** ensures that changes made to the source code has not created defects in the existing source code.
- ▶ **Installation testing:** checks how well the software application works on the required platform.

- ▶ **Robustness testing:** establish the ability of the software application in handling anomalies (being *trustworthy*).
- ▶ **Performance testing:** check if the system is fast enough and it uses acceptable amount of memory (e.g., is under stress).

- ▶ **Level 1:** testing modules (i.e., classes), then program (i.e., use cases), then suites (i.e., application).
- ▶ **Level 2 (*Alpha Testing or Verification*):** Execute programs in a simulated environment and test inputs and outputs.
- ▶ **Level 3 (*Beta Testing or Validation*):** test in a live use environment and test for response times, performance under load and recovery from failure, etc.

Summary of Testing Activities

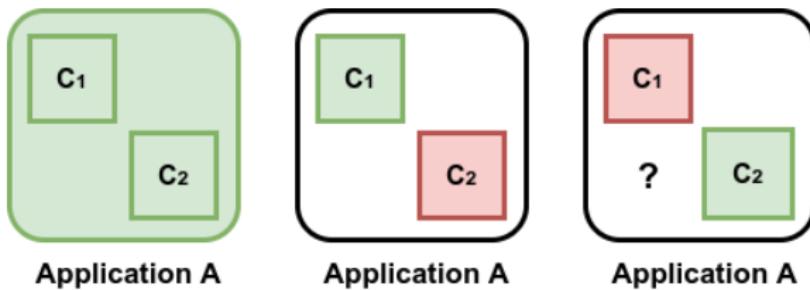


Braude & Bernstein (2011)

- ▶ Regression testing is a process in which the software is **retested** so as to ensure that its behaviour has not been compromised by the addition of new, or change to existing code.
- ▶ It facilitates faster development.
 - ▶ It reduces risk of changes.
 - ▶ Unexpected effects can be found quickly.
 - ▶ Ability to run thousands of tests in one click.

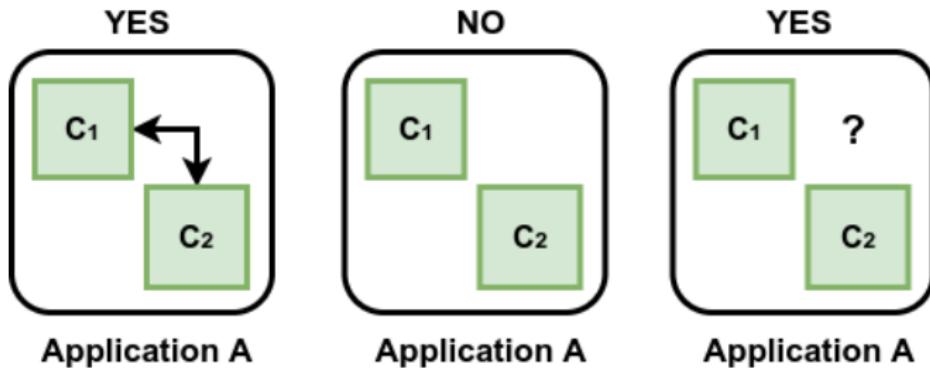
When is Regression Testing needed?

- ▶ Regression testing is important, but can be very time and cost consuming.
 - ▶ Requires capturing of data, analysis, reporting, etc.
- ▶ Consider the following scenario:



- ▶ When should regression testing be performed?

- When should regression testing be performed?



- If C_1 depends on C_2 , then Yes.
- If C_1 does not depend on C_2 , then No.
- If unsure, YES!

- ▶ **Test plans**
 - ▶ Written before the tests are carried out!
 - ▶ Written, in fact, before the code is written.
 - ▶ Contain test cases.
- ▶ **Test cases** are specified with the following formation:
 - ▶ Test data,
 - ▶ expected outcomes,
 - ▶ description of the test,
 - ▶ test environment and configuration.

► Test results

- ▶ Used to facilitate analysis, ideally test results should be recorded in a spreadsheet, database or a specialist package (e.g., Jira).
- ▶ Record when tests are failed or passed.
- ▶ Allow reporting of percentage passed.
- ▶ Ideally should be linked to requirements, showing whether or not each requirement has been met.
- ▶ Error results should be recorded in a fault reporting package, with enough details for developers to **reproduce** them with a view to fixing the bugs.

1. Define what is meant by a “testing unit”.
2. Determine the *types* of testing to be performed.
3. Determine the *extent* of testing (i.e., prioritise the tests).
4. Determine how to document the testing procedures.
5. Determine *input sources*.
6. Decide who will perform which tests.
7. Estimate *resources*.
8. Identify *metrics* to be collected.

- ▶ Software should be thoroughly tested before *release*, but when should testing stop?
- ▶ There is no “*one size fits all*” scheme, in software testing.
- ▶ The testing team should establish a set of stopping criteria:
 - ▶ A tester is unable to find another defect in **n** minutes of testing (where n may be any +ve integer, e.g., 5, 10, 30, 100).
 - ▶ When all **nominal, boundary** and **out-of-bounds** test data show no defect.
 - ▶ When a given **checklist** of test types has been completed.
 - ▶ When testing runs out of its scheduled time (**DANGEROUS**).

- ▶ **Unit testing** is performed to test parts (classes and their methods) of an application *in isolation*.
- ▶ White box unit testing:
 - ▶ **Code coverage**: test cases cause every line of code (*statement*), all conditions (*branch*) and all possible flows (*path*), from entry to exit, to be executed.
- ▶ Black box unit testing:
 - ▶ **Equivalence partitioning**: divide input values into equivalent groups.
 - ▶ **Boundary value analysis**: test at boundary conditions.

- ▶ Performing **path coverage** on the following method:

```
1 private static void showResult(int guessed, int toGuess) {  
2     if (guessed == toGuess) {  
3         System.out.println("Well done! You guessed it!");  
4     }  
5     else if (guessed > toGuess) {  
6         System.out.println("Sorry. Your guess is too high.");  
7     }  
8     else {  
9         System.out.println("Sorry. Your guess is too low.");  
10    }  
11    System.out.println("The number I have in mind is %d\n", toGuess);  
12 }
```

- ▶ A test case for each **distinct path** is:

- ▶ showResult (8, 8): 2 → 3 → 11
- ▶ showResult (9, 8): 2 → 5 → 6 → 11
- ▶ showResult (7, 8): 2 → 5 → 8 → 9 → 11

- ▶ **Assertions** are statements in Java which ensure the correctness of assumptions.
- ▶ Allow us to enforce business rules and catch any mistakes, errors.
- ▶ Allow us to check for parameter and return values.

```
1 public void checkCanDrive(int age) {  
2     assert age <= 17 : "Cannot_drive";  
3     System.out.println("Can_drive_at_" + age);  
4 }
```

- ▶ Generally, they allow us to check for something that should never happen.

- ▶ **JUnit** is a framework for unit testing which facilitates the writing of repeatable tests.
- ▶ To use JUnit 5, typically one imports:
 - ▶ org.junit.jupiter.api.Assertions
 - ▶ org.junit.jupiter.api.Test
- ▶ Annotates a test method with a @Test
- ▶ Uses assert- methods, i.e., assertFalse, assertTrue, assertEquals, assertThrows, etc.

```
1  @Test
2  public void testAddingStaff() {
3      StaffManager sm = new StaffManager();
4      sm.addStaff("Lucy", "Bastin");
5      Assertions.assertFalse(sm.getAllStaff().isEmpty());
6      Assertions.assertEquals(1, sm.getAllStaff().size());
7  }
```

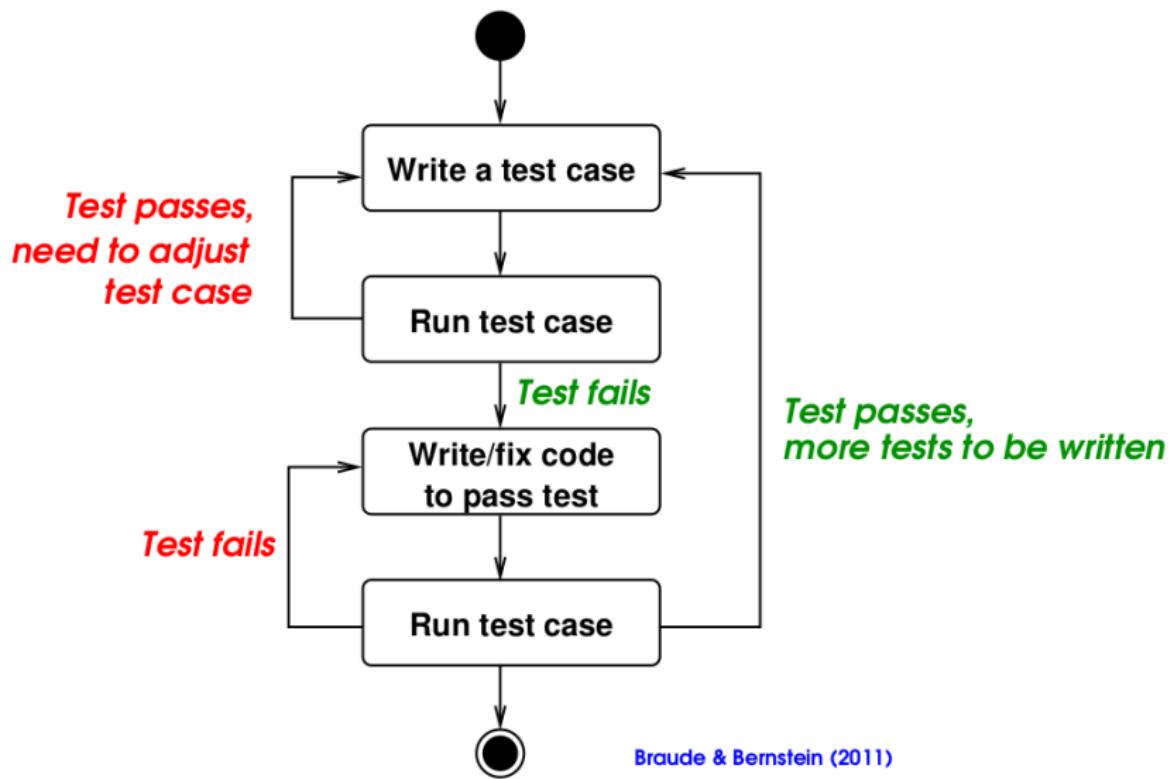
- ▶ **Fixtures** allow you to set up a testing environment: prepare the baseline of your tests.
- ▶ Usually fixtures have the following lifecycle:

```
1 @BeforeAll
2 @BeforeEach
3 @Test
4 @AfterEach
5 @AfterAll
```

- ▶ You may want to use the “Before” annotations for the preparation of test data, and the “After” annotations for housekeeping.

<https://junit.org/junit5/docs/current/user-guide/>

- ▶ A different approach to writing software.
- ▶ The development of the application is thus driven by the test cases solely.
- ▶ Writing test cases *before* writing any source code.
- ▶ Code that does not address the issues in the test cases will not be written.
- ▶ A testing framework such as JUnit can be used to facilitate TDD.



Braude & Bernstein (2011)

- ▶ Test cases ensure a good statement coverage.
- ▶ Only code required to address the requirements is written; no redundancy.
- ▶ Rapid feedback on the written code.
- ▶ Test suite available for further testing purposes, e.g., regression testing.

Today's learning outcomes:

- ▶ Software Testing.
- ▶ White and black box testing.
- ▶ Unit Testing and JUnit.
- ▶ Test-Driven Development.

Unit 7 Software Design: an introduction

Unit Outcomes. Here you will learn

- ▶ to identify the difference between analysis and design
- ▶ to identify the difference between logical and physical design, and system and detailed design
- ▶ to describe characteristics of a good design
- ▶ to recognise some of the characteristics of trustworthy software
- ▶ to appreciate the need to make trade-offs in design

Further Reading: Bennett, et al. (2010)

Ch12

Contents

- ▶ Design vs. Analysis
- ▶ Analysis and design in project management
- ▶ Logical vs. Physical Design
- ▶ System design and detailed design
- ▶ Qualities of good design
- ▶ Design tradeoffs
- ▶ Trustworthy software
- ▶ Measurable Objectives in Design

How to use these slides

- ▶ The slides are very complete.
- ▶ They contain a number of additional examples and references
- ▶ This is designed to direct and support your independent study.
- ▶ In the lecture, we will skip over some of this detail.
- ▶ Please review the full slides in your own time, and ask any questions on the discussion board

How is design different from analysis? In a nutshell... .

- ▶ *Analysis* identifies “*what*” the system must do.
- ▶ *Design* specifies “*how*” it will do it.
- ▶ In *analysis*, the analyst seeks to *understand*:
 - ▶ the organization,
 - ▶ its requirements, and
 - ▶ its objectives.
- ▶ In *design*, the designer seeks to *specify* a system that will:
 - ▶ fit the organization,
 - ▶ meet its requirements adequately, and
 - ▶ assist it to meet its objectives.

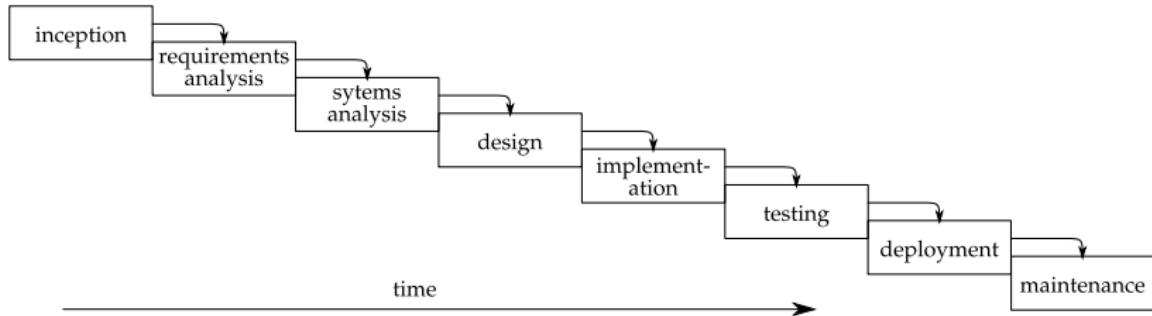
How is design different from analysis? In a nutshell... .

Consider a computer system which supports the day-to-day work of the *advertising company Agate*:

- ▶ *Analysis* identifies that there needs to be an (*advertising*) Campaign class which has a `title` attribute.
 - ▶ *Design* determines:
 - ▶ how the `title` attribute will be *entered* into the system,
 - ▶ how the `title` attribute will be *displayed* on screen,
 - ▶ how the `title` attribute will be *stored* in a database,
 - ▶ what other attributes of the class Campaign are, and
 - ▶ other classes, etc.
- For more info about the **Agate** case study, see Bennett et al. (2010).

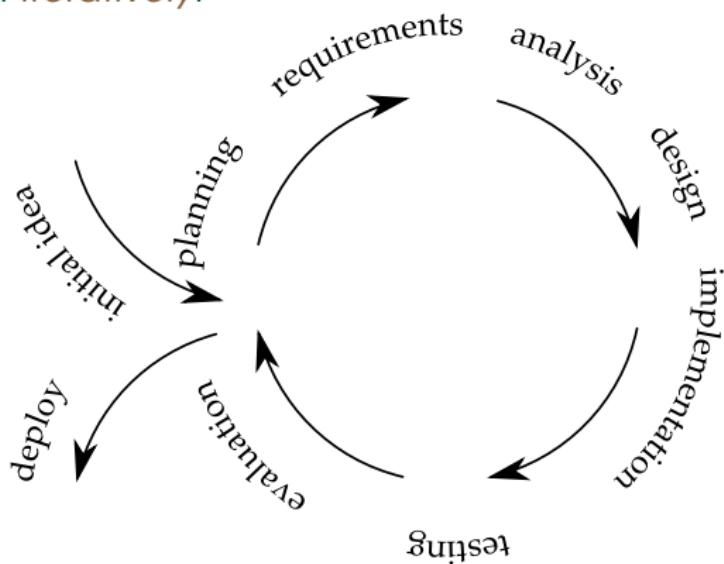
When does analysis stop and design start?

- In a *waterfall life cycle*, there is a *clear transition* between *analysis* and *design*.
 - Activities in one stage (analysis, design, construction, etc.) is completed *before* activities in the next stage begins.



When does analysis stop and design start?

- In an *iterative life cycle*, the analysis of a particular part of the system will precede its design, but analysis and design may be happening in parallel.
 - Analysis and design happen throughout the *entire* project development *iteratively*.

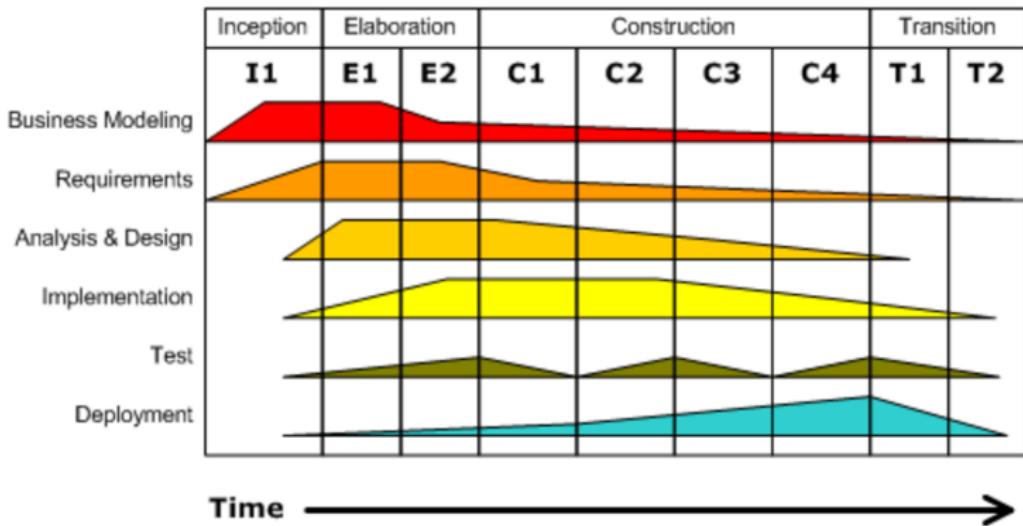


When does analysis stop and design start?

- In the *Unified Process*, the amount of analysis and design performed *grows* from the *Inception* phase to the *Elaboration* phase and *shrinks* in the *Construction* phase.

Iterative Development

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



Logical and Physical Design

How do they differ?

- ▶ *Logical design* is *independent* of the implementation language and platform.
- ▶ *Physical design* is based on the *actual* implementation platform and the language that will be used.

Logical and Physical Design

How do they differ?

- ▶ e.g. Logical design of a UI can be done without knowing the implementation language.
 - types of fields, position in windows, etc.
- ▶ Physical design is specific to the language.
 - E.g. layout managers available, etc.
- ▶ Separating logical / physical design is useful if the software must be implemented on *different platforms*.
- ▶ A *platform-independent design* that can be tailored as required.

System Design

- ▶ *System design* deals with the high level **architecture** of the system:
 - ▶ structure of sub-systems, communication between them, and their distribution on processors
 - ▶ standards for screens, reports, help, etc.

- ▶ Traditional *detailed design* consists of:
 - ▶ designing inputs, outputs, processes and file/database structures
- ▶ Object-oriented **detailed design** adds detail to the analysis model, e.g.:
 - ▶ types of attributes
 - ▶ operation signatures
 - ▶ additional classes to handle user interface or data management
 - ▶ design of reusable components
 - ▶ assigning classes to packages

Qualities of Design

(1)

- ▶ *Functional*: the system will perform its required functions
- ▶ *Efficient*: the system performs those functions efficiently in terms of *time* and *resources*
- ▶ *Economical*: running costs of system will not be unnecessarily high
- ▶ *Reliable*: not prone to hardware or software failure, will deliver the functionality when the users want it
- ▶ *Secure*: protected against errors, attacks and loss of valuable data
- ▶ *Flexible*: capable of being adapted to new uses, to run in different countries or to be moved to a different platform

Qualities of Design

(2)

- ▶ *General*: general-purpose and portable (mainly applies to utility programs)
- ▶ *Buildable*: the design is not too complex for the developers to be able to implement it
- ▶ *Manageable*: easy to estimate work involved and to check on progress
- ▶ *Maintainable*: the design makes it possible for the maintenance programmer to understand the designer's intention
- ▶ *Usable*: provides users with a satisfying experience (not a source of dissatisfaction)
- ▶ *Reusable*: elements of the system can be reused in other

Prioritizing Design Trade-offs

- ▶ A designer is often faced with objectives that are mutually *incompatible*.
- ▶ **Trade-offs** have to be applied to resolve these conflicts.
- ▶ *Functionality*, *reliability* and *security* are likely to conflict with *economy*.
 - E.g. the number of *features* in the end-product is constrained by the budget available for the development of the system.
- ▶ It is helpful if *guidelines* are prepared for *prioritizing* design objectives.
- ▶ If design choice is *unclear*, users should be consulted.

Trade-offs in Design

- ▶ Design objectives may conflict with *constraints* imposed by requirements.
- ▶ An example:

Requirement: The system can be used in different countries by speakers of different languages.

Entails: Designers will have to agree a list of all prompts, labels and messages and refer to these by some system of naming or numbering.

Trade-offs: Multilingual support will lead to increases in *flexibility and maintainability*, but also increase the *cost of design*.

Question: Is the above requirement functional or non-functional?
Why?

Trustworthy Software

A Case Study: MCAS on Boeing's 737 Max

Trustworthiness seeks to address the *quality* and *robustness* of software, ensuring that the software "*performs as it should, when it should and how it should*" (UK-TSI, 2016).

See Blackboard (Unit 7) for a video illustration about MCAS from BBC News at <https://web.archive.org/web/20230107001748/https://www.bbc.co.uk/news/extra/IFTb42kkNv/boeing-two-deadly-crashes>

For a more current example, check out the BBC Radio 4 investigation of the Post Office Horizon software scandal at
<https://www.bbc.co.uk/programmes/m001sd41>

Trustworthy Software

BS 10754-1:2018 Systems Trustworthiness

- ▶ With the pervasiveness of software in our daily life, it is increasingly important to ensure that the software we use is trustworthy.
- ▶ *BS 10754-1:2018 Systems Trustworthiness* is a publicly available specification hosted by the *British Standard Institution (BSI)*.
- ▶ It gives guidance on procedures by which organisations procure or supply trustworthy software.

Standard

Information technology. Systems trustworthiness - Governance and management specification

BS 10754-1:2018 • Current • 28 Feb 2018



Overview



Preview



Product Details

Preview Document Contents

Purchase Options



Digital



Hard copy



Non-Member Total

£218.00

Member Total

£109.00

Save up to 50% on this Standard by becoming a member.



 Add to Basket

BS 10754-1:2018 Systems Trustworthiness

Available through the Library under our license - encrypted document so you need to install the FileOpen application to read it.

BS 10754-1:2018



BSI Standards Publication

Information technology — Systems trustworthiness

Trustworthy Software

Trustworthy Software Foundation Guidance

- ▶ The Trustworthy Software Foundation (TSF) summarises five facets of trustworthiness:
 - ▶ ***Safety:** The ability of the software to operate without causing harm to anything or anyone.*
 - ▶ ***Reliability:** The ability of the software to operate correctly.*
 - ▶ ***Availability:** The ability of the software to operate when required.*
 - ▶ ***Resilience:** The ability of the software to recover from errors quickly and completely.*
 - ▶ ***Security:** The ability of the software to remain protected against the hazards posed by malware, hackers or accidental misuse.*

Trustworthy Software

Trustworthy Software Essentials

- ▶ The guidance document (UK-TSI, 2016) "*sets out the baseline requirements for managing trustworthiness during the software life-cycle*":
 - ▶ Scope for Use (E1)
 - ▶ Coding Practices (E2)
 - ▶ Use Tools Effectively (E3)
 - ▶ Defect Management (E4)
 - ▶ Artefact Management (E5)
 - ⇒ Many of them are well established good software engineering practice.

<http://tsfdn.org/wp-content/uploads/2016/03/TS502-1-TS-Essentials-Guidance-Issue-1.2-WHITE.pdf>

Achieving Measurable Objectives in Design

- ▶ Design addresses the issue of *how* to meet the requirements.
- ▶ **Non-functional requirements**, in particular, impact on the design.
- ▶ **Measurable objectives** often refer to the *non-functional requirements* of a software development project.
 - Measurable objectives set clear *targets* for designers.
- ▶ Objectives should be formulated in a *quantifiable* manner so that they can be tested.

Achieving Measurable Objectives in Design

- ▶ In the context of software design, which of the following measurable objectives are **verifiable**?
 1. *To reduce invoice errors by one-third within a year.*
 2. *To process 50% more orders at peak periods.*
 3. *To improve the sales of **Mars Bar** by 15%.*
 4. *To reduce the turnaround time of feedback to coursework submissions.*
 5. *To improve the quality of feedback to students.*
 6. *To increase the number of properties managed by a property manager by 25%.*

References

- ▶ Bennett, S., McRobb, S. and Farmer, R., *Object-Oriented Systems Analysis and Design Using UML*, 4th ed, Maidenhead: McGraw-Hill, 2010.
- ▶ Braude, E.J. and Bernstein, M.E., *Software Engineering: Modern Approaches*, 2nd ed, Hoboken, NJ: Wiley, 2011.
- ▶ Jacobson, I., Booch, G. and Rumbaugh, J., *The Unified Software Development Process*, Reading, MA: Addison-Wesley; ACM Press, 1999.
- ▶ Kruchten, P., *The Rational Unified Process: An Introduction*, Reading, MA: Addison-Wesley, 2004.
- ▶ Lunn, K., *Software Development with UML*, London: Palgrave Macmillan, 2003.

References

- ▶ Revell, A., *Trustworthy software*, (Online). Available at: <https://www.bcs.org/content-hub/trustworthy-software/> (14/10/2019).

- ▶ Rumbaugh, J., 'Models through the development process', *Journal of Object-Oriented Programming*, May, 1997.

References

- UK-TSI, *Trustworthy Software Guidance Document: Trustworthy Software Essentials (TSE)*, TS502-1, Traffic Light Protocol (TLP) White, Issue 1.2, February 2016. (Online). Available at: <http://tsfdn.org/wp-content/uploads/2016/03/TS502-1-TS-Essentials-Guidance-Issue-1.2-WHITE.pdf> (7/11/2021).

See Chapter 12 of Bennett et al. (2010) for detail.

Learning Outcomes

Learning Outcomes. You should now be able to

- ▶ state the difference between analysis and design
- ▶ state the difference between logical and physical design
- ▶ state the difference between system and detailed design
- ▶ describe some software engineering practice that would promote software trustworthiness
- ▶ evaluate the trustworthiness of a piece of software
- ▶ evaluate if a software exhibits qualities of a good design
- ▶ take into account of the need for trade-offs when designing software

Unit 8 Software Architecture

Unit Outcomes. Here you will learn

- ▶ what is meant by architecture in information systems development
- ▶ various architectural styles, including layers, MVC and SOA
- ▶ how to apply the Model-View-Controller (MVC) architecture

Contents

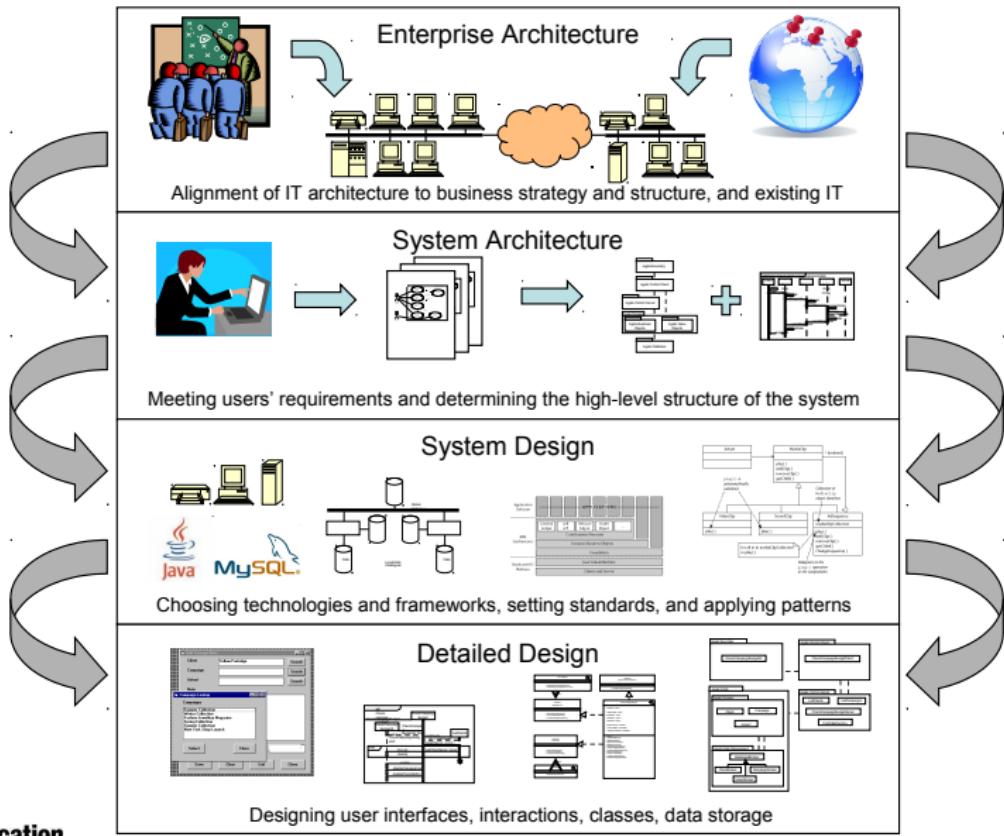
- ▶ Relationship between architecture and design
- ▶ Enterprise, System and Software Architecture
- ▶ Subsystems
 - ▶ Layering v Partitioning
 - ▶ Layers v. Tiers
- ▶ Model-View-Controller (MVC)
- ▶ Communication between subsystems
 - ▶ Peer-to-peer, Client-server
- ▶ Service-Oriented Architecture (SOA)

Architecture

An Architecture... (Eeles, 2006)

- ▶ defines structure
 - What are the core components, and how do they relate to each other?
- ▶ defines behaviour
 - E.g. How do the core components interact with each other?
- ▶ balances stakeholders' needs which may conflict
- ▶ has a particular scope
 - E.g. enterprise, system, software, hardware, organisational, information

Relationship between Architecture and Design



Enterprise Architecture

Bennett et al. (2010, Section 13.4.2)

- ▶ **Enterprise architecture** concerns:
 - ▶ the modelling of the *business*,
 - ▶ the way the *enterprise* conducts business and
 - ▶ how the *information systems* are intended to support the business.
- ▶ *Typical questions* to be considered include:
 - ▶ How does the system overlap with other systems in the organisation?
 - ▶ How will the system need to interface with other systems?
 - ▶ Will the system help the organisation to achieve its goals?
 - ▶ Is the cost of the system justified?

System Architecture

Key Definitions : Garland & Anthony (2003) and IEEE (2000)

- ▶ *System* is a set of *co-operating components* organised to accomplish a specific function or a set of functions.
 - ➡ System architecture describes the hardware and software elements and interactions between them.

System Architecture

System architects:

- ▶ act on behalf of the client;
- ▶ address the *big picture*;
- ▶ ensure that the required *qualities* of the system are accounted for in the design;
- ▶ ensure the required features are provided at the right cost.

Software Architecture

What is it?

- ▶ “A *software architecture* describes the overall components of an application and how they relate to each other. Its design goals, include sufficiency, understandability, modularity, high cohesion, low coupling, robustness, flexibility, reusability, efficiency, and reliability.”

Braude and Bernstein (2011)

Software Architecture

Architecture ≠ Framework

- ▶ The terms *architecture* and *framework* have sometimes been used interchangeably, but that is *inappropriate*.
- ▶ *Architecture ≠ Framework*
- ▶ "A *framework*, sometimes called a *library*, is a collection of software artifacts usable by several different applications. These artifacts are typically implemented as classes, together with the software required to utilize them. A framework is a kind of common denominator for a family of applications. The Java APIs (3D, 2D, Swing, etc.) are frameworks."

Braude and Bernstein (2011, p.358)

Framework : Examples

Other examples of frameworks:

- ▶ *Java*: Apache Struts, Spring, JMF, JavaFX...
- ▶ *PHP*: Laravel, Drupal, CakePHP...
- ▶ *python*: Django, Dash, Giotto...
- ▶ *Javascript, HTML, CSS*: JQuery, Bootstrap, React...

Subsystems

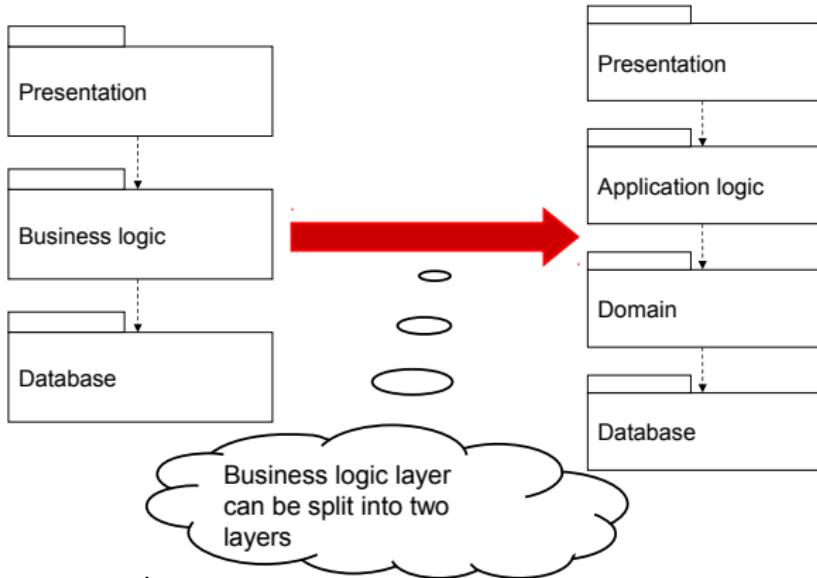
- ▶ A *subsystem* typically groups together elements of the system that share some *common properties*.
- ▶ The subdivision of an information system into subsystems has the following *advantages*:
 - ▶ It produces *smaller units* of development.
 - ▶ It helps to *maximize reuse* at the component level.
 - ▶ It helps the developers to *cope with complexity*.
 - ▶ It improves *Maintainability* and *Portability*.

Layering and Partitioning

- ▶ Two general approaches to divide a software system into subsystems:
 - ▶ *Layering*: so called because the different subsystems usually represent different *levels of abstraction*.
 - ▶ *Partitioning*: usually means that each subsystem focuses on a different aspect of the *functionality* of the system as a whole.
- ▶ Both approaches are often used together on one system.

Layered Architecture

Simple Layered Architecture



Layered Architecture

Simple Layered Architecture

- ▶ *Boundary* classes can be mapped onto the *presentation* layer. (e.g. end-user interaction, like forms)
- ▶ *Control* classes can be mapped onto the *application logic* layer. (e.g. I/O, processing inputs)
- ▶ *Entity* classes can be mapped onto a *domain* layer. (e.g. data model)
- ▶ *Database* layer corresponds to the storage of data within the system. (physical storage)

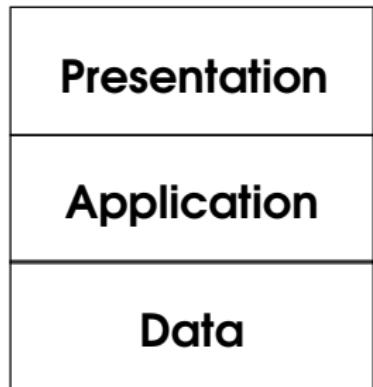
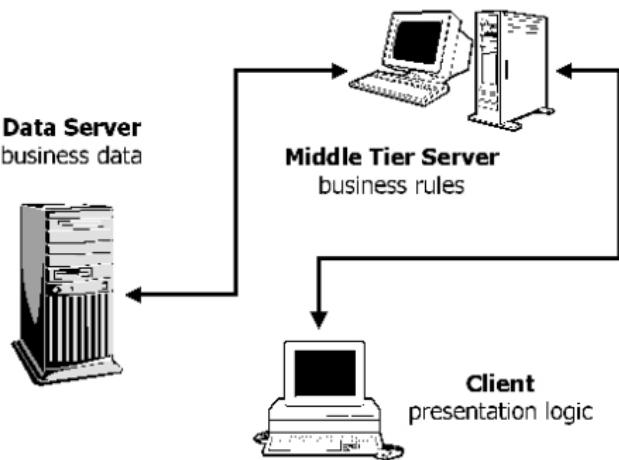
Tiered Architecture

Layered architectures Vs Tiered architectures

- ▶ A *layered architecture* - *logical* structuring:
 - ▶ groups functionality / code based on its level of abstraction.
 - ▶ All layers may reside on the *same* computer.
- ▶ A *tiered architecture* - *physical* structuring:
 - ▶ concerns the logical grouping of functionality / code, BUT
 - ▶ Tiers reside on, and are executed by, *different* computers.

Tiered Architecture

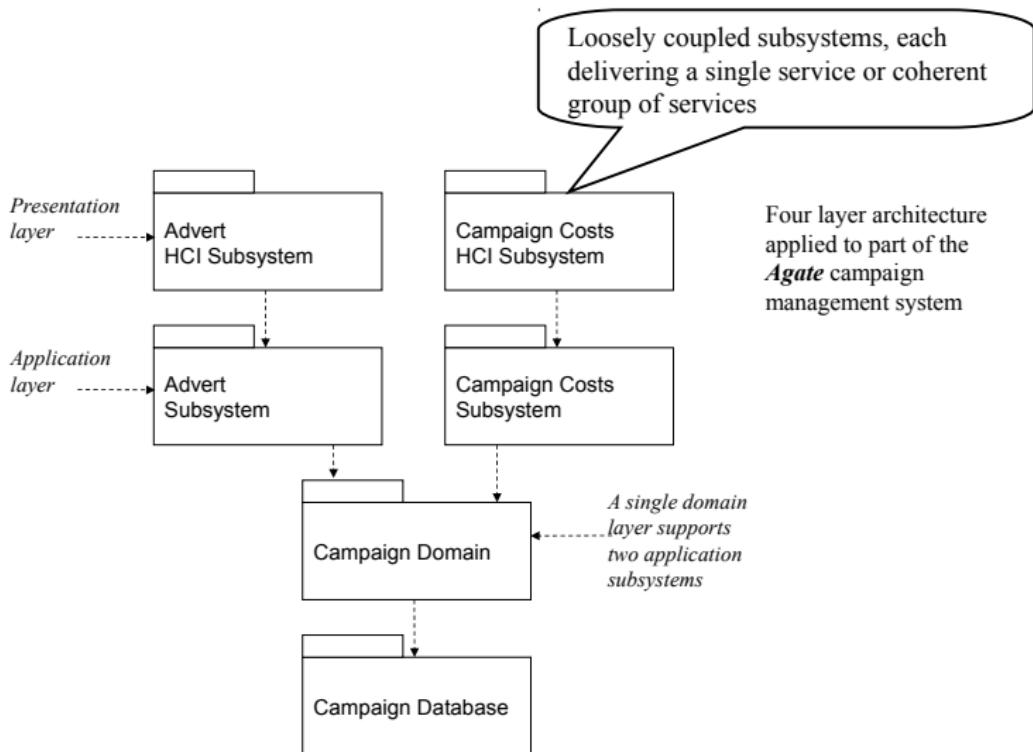
Three-tier Architecture (Ramirez, 2000)



Source:

<http://www.linuxjournal.com/files/linuxjournal.com/linuxjournal/articles/035/3508/3508f3.jpg>

Partitioning Partitioned Subsystems



But...

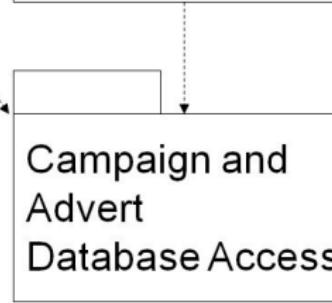
An issue with some architectures

Multiple interfaces for the same core functionality.

*Each subsystem
contains some
core functionality*



*Changes to data in
one subsystem need to be
propagated to the others*



But...

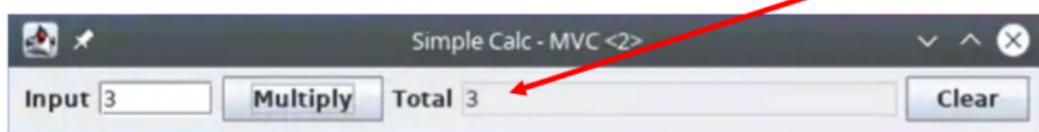
An issue with some architectures : Difficulties

- ▶ The *same information* should be capable of being presented in *different formats* in different windows.
- ▶ Changes made within one view should be *reflected immediately* in the other views.
 - ➡ ILLUSTRATION: A simple multiplier...

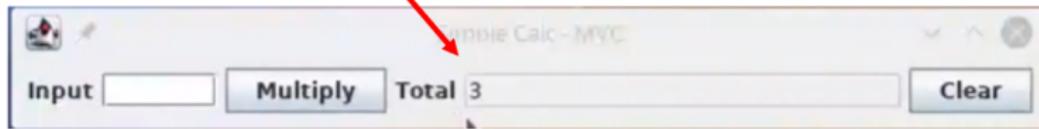
A simple calculator in MVC

In View 1, the user inputs the number '3'.

This is used to multiply the stored total of 1, and the new value '3' is shown in the view.

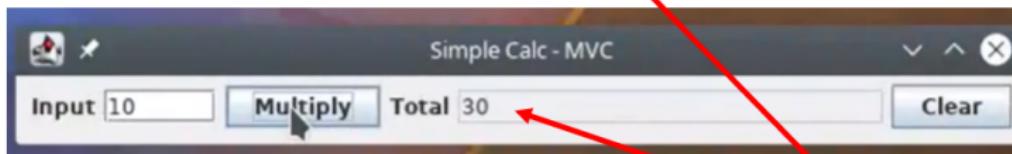
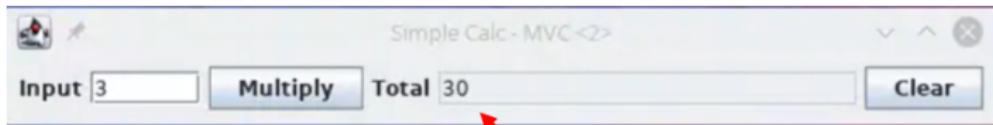


The independent view 2 also has its total updated to 3, so that the user knows what value is currently in the shared store



Homework: watch the video on Blackboard (Dr. S. Wong)

A simple calculator in MVC



In View 2, the user inputs the number '10'.

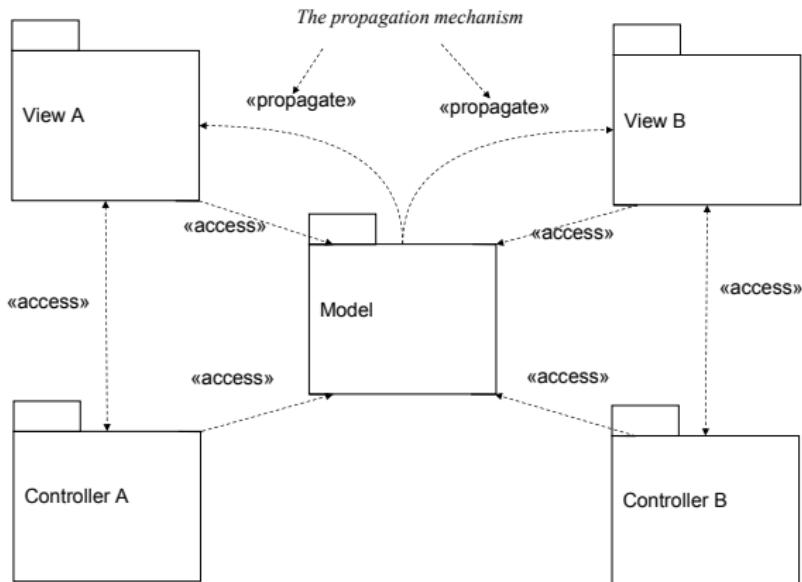
This is used to multiply the stored total of 3, and the new value '30' is shown in BOTH views.

Without this dynamic update, neither user would understand why their totals kept jumping illogically

Model-View-Controller (MVC)

Bennett et al. (2010)

- ▶ *Propagation Mechanism*: enables the model to inform each view that the model data has changed and as a result the view must update itself.
It is also often called the *dependency mechanism*.

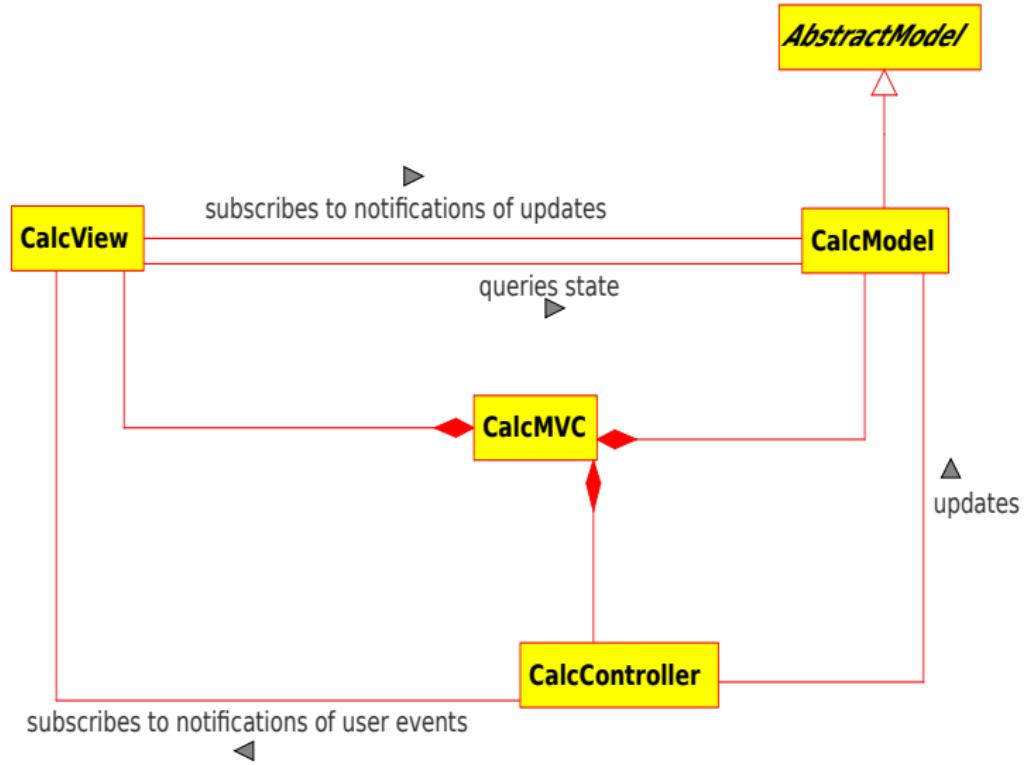


Model-View-Controller (MVC)

Eckstein (2007)

- ▶ **Model**: The model represents *data* and the rules that govern access to and updates of this data.
- ▶ **View**: The view *renders* the contents of a model and must update its presentation as needed. The view may *register* itself with the model for change notifications, or may *call* the model to retrieve the most current data.
- ▶ **Controller**: The controller *translates* the user's interactions with the view into actions that the model will perform.
In a stand-alone GUI client, user interactions could be button clicks or menu selections, whereas in an enterprise web application, they appear as **GET** and **POST** HTTP requests.

Model-View-Controller (MVC) UML Class Diagram



Model-View-Controller (MVC)

Roles of each component

- ▶ CalcMVC:
 - ▶ is the *top-level* class.
 - ▶ responsible for *creating* the **model**, **view** and **controller** objects.
- ▶ CalcModel:
 - ▶ does *not know* about the **view** nor the **controller**.
 - ▶ *fires* property change notifications to its subscribers, in this case, the **view**.

Model-View-Controller (MVC)

Roles of each component

- ▶ CalcView:

- ▶ *renders* the **model** using a GUI.
- ▶ *subscribes* to the property change notifications from the **model**.
- ▶ *passes* user input events to its subscribers.

- ▶ CalcController:

- ▶ *subscribes* to all user input events.
- ▶ *defines* the event handlers for GUI events.
- ▶ *calls* the **model** to change its state based on user event.

CalcMVC (Swartz, 2004)



```
1 // structure/calc-mvc/CalcMVC.java - Calculator in MVC pattern.  
2 // Fred Swartz - December 2004  
3  
4 import javax.swing.*;  
5  
6 public class CalcMVC {  
7     //... Create model, view, and controller.  
8     public static void main(String[] args) {  
9  
10         CalcModel      model      = new CalcModel();  
11         CalcView       view       = new CalcView(model);  
12         CalcController controller = new CalcController(model, view);  
13  
14         view.setVisible(true);  
15     }  
16 }
```

Source: <http://leepoint.net/notes-java/GUI/structure/40mvc.html>

CalcModel (1 of 3)

```
1 // structure/calc-mvc/CalcModel.java
2 // Fred Swartz - December 2004
3 // Modified by M Konecny 21-01-2011
4 // Model
5 // This model is completely independent of the user interface.
6 // It could as easily be used by a command line or web interface.
7
8 import java.math.BigInteger;
9
10 public class CalcModel extends AbstractModel {
11     /** name used by property change notification mechanism */
12     public static final String TOTAL_PROPERTY = "Total";
13     private static final String INITIAL_VALUE = "1";
14
15     private BigInteger m_total; // The total current value state.
16
17     /** Constructor */
18     CalcModel() {
19         reset();
20     }
```

CalcModel (2 of 3)

```
1  /** Reset total to initial value. */
2  public void reset() {
3      BigInteger old_value = m_total;
4      m_total = new BigInteger(INITIAL_VALUE);
5      firePropertyChange(TOTAL_PROPERTY, old_value, m_total);
6  }
7
8  /**
9   * Multiply current total by a number.
10  * @param operand
11  *         Number (as string) to multiply total by.
12  */
13 public void multiplyBy(BigInteger operand) {
14     BigInteger old_value = m_total;
15     m_total = m_total.multiply(operand);
16     firePropertyChange(TOTAL_PROPERTY, old_value, m_total);
17 }
```

CalcModel (3 of 3)

```
1  /**
2   * Set the total value.
3   * @param value
4   *          New value that should be used for the calculator total.
5   */
6  public void setValue(BigInteger value) {
7      BigInteger old_value = m_total;
8      m_total = value;
9      firePropertyChange(TOTAL_PROPERTY, old_value, m_total);
10 }
11
12 /**
13  * Return current calculator total. */
14 public BigInteger getValue() {
15     return m_total;
16 }
```

AbstractModel (1 of 2)

```
1 import java.beans.PropertyChangeListener;
2 import java.beans.PropertyChangeSupport;
3
4 /**
5  * This class provides base level functionality for all models,
6  * including a support for a property change mechanism (using
7  * the PropertyChangeSupport class), as well as a convenience
8  * method that other objects can use to reset model state.
9 *
10 * @author Robert Eckstein
11 * from: http://www.oracle.com/technetwork/articles/javase/mvc-136693.html
12 */
13 public abstract class AbstractModel {
14
15     /**
16      * Convenience class that allow others to observe changes
17      * to the model properties
18      */
19     protected PropertyChangeSupport propertyChangeSupport;
20
21     /** Default constructor: Instantiates class PropertyChangeSupport. */
22     public AbstractModel() {
23         propertyChangeSupport = new PropertyChangeSupport(this);
24     }
}
```

AbstractModel (2 of 2)

```
1  /** Adds a property change listener to the observer list. */
2  public void addPropertyChangeListener(PropertyChangeListener l) {
3      propertyChangeSupport.addPropertyChangeListener(l);
4  }
5
6  /** Removes a property change listener from the observer list. */
7  public void removePropertyChangeListener(PropertyChangeListener l) {
8      propertyChangeSupport.removePropertyChangeListener(l);
9  }
10
11 /**
12  * Fires an event to all registered listeners informing them
13  * that a property in this model has changed.
14  */
15 protected void firePropertyChange(String propertyName,
16         Object oldValue, Object newValue) {
17     propertyChangeSupport.firePropertyChange(propertyName,
18         oldValue, newValue);
19 }
20 }
```

CalcView (1 of 2)

```
1 // structure/calc-mvc/CalcView.java - View component
2 // Presentation only. No user actions.
3 // Fred Swartz - December 2004
4 // Modified by M Konecny 21-01-2011
5
6 import java.awt.event.*;
7 import java.beans.PropertyChangeEvent;
8 import java.beans.PropertyChangeListener;
9 // other import statements omitted
10 public class CalcView extends JFrame {
11
12     private CalcModel m_model;
13     // other field definitions and initialisation omitted.
14
15     /** Constructor */
16     CalcView(CalcModel model) {
17         // ... Set up the logic
18         m_model = model;
19         // ... Initialize components
20         m_totalTf.setText(m_model.getValue().toString());
21         m_totalTf.setEditable(false);

```

CalcView (2 of 2)

```
1     // ... Setup automatic updates of the total from model:  
2     m_model.addPropertyChangeListener(new PropertyChangeListener() {  
3         // will be executed by the model when an event is fired there.  
4         public void propertyChange(PropertyChangeEvent evt) {  
5             if (evt.getPropertyName().equals(m_model.TOTAL_PROPERTY)) {  
6                 m_totalTf.setText(evt.getNewValue().toString());  
7             }  
8         }  
9     });  
10  
11     // other GUI code omitted  
12 }  
13  
14 // other GUI methods omitted  
15  
16 void addMultiplyListener(ActionListener mal) {  
17     m_multiplyBtn.addActionListener(mal);  
18 }  
19  
20 void addClearListener(ActionListener cal) {  
21     m_clearBtn.addActionListener(cal);  
22 }  
23 }
```

CalcController (1 of 3)

```
1 // structure/calc-mvc/CalcController.java - Controller
2 // Handles user interaction with listeners.
3 // Calls View and Model as needed.
4 // Fred Swartz - December 2004, modified by M Konecny 21-01-2011
5
6 import java.awt.event.*;
7 public class CalcController {
8     // The Controller needs to interact with both the Model & View.
9     private CalcModel m_model;
10    private CalcView m_view;
11
12    /** Constructor */
13    CalcController(CalcModel model, CalcView view) {
14        m_model = model;
15        m_view = view;
16        //... Add listeners to the view.
17        view.addMultiplyListener(new MultiplyListener());
18        view.addClearListener(new ClearListener());
19    }
}
```

CalcController (2 of 3)

```
1 ////////////// inner class MultiplyListener
2 /** When a multiplication is requested.
3 * 1. Get the user input number from the View.
4 * 2. Call the model to multiply by this number.
5 * If there was an error, do nothing.
6 */
7 class MultiplyListener implements ActionListener {
8     public void actionPerformed(ActionEvent event) {
9         try {
10             m_model.multiplyBy(m_view.getUserInput());
11
12         } catch (Exception e) {
13             // ignore the exception, ie ignore the event
14         }
15     }
16 } //end inner class MultiplyListener
```

CalcController (3 of 3)

```
1      /////////////////// inner class ClearListener
2  /**  Reset model. (View will update automatically)
3   */
4
5  class ClearListener implements ActionListener {
6      public void actionPerformed(ActionEvent e) {
7          m_model.reset();
8      }
9  } // end inner class ClearListener
10 } // end of class CalcController
```

Model-View-Controller (MVC)

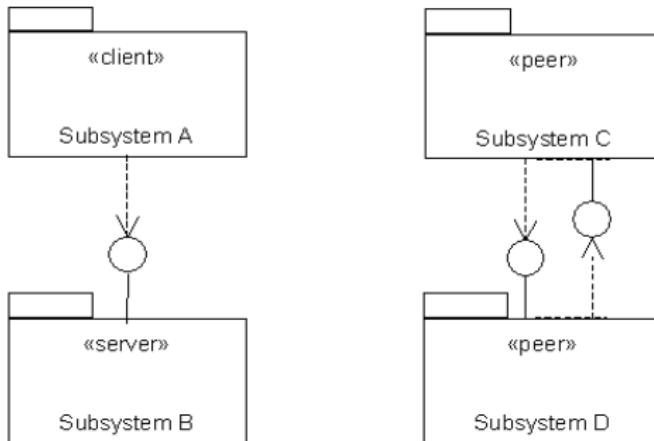
Single Model, Multiple Views

- ▶ The MVC architecture is *flexible*.
- ▶ *Multiple* views can be created in the MVC architecture.
- ▶ Changes made to the model within one view is *reflected immediately* to *all* views which use the same model.

```
1 import javax.swing.*;
2 public class CalcMVC {
3     public static void main(String[] args) {
4         CalcModel      model      = new CalcModel();
5         CalcView       view1      = new CalcView(model);
6         CalcController controller1 = new CalcController(model, view1);
7         CalcView       view2      = new CalcView(model);
8         CalcController controller2 = new CalcController(model, view2);
9
10        view1.setVisible(true);
11        view2.setVisible(true);
12    }
13 }
```

Subsystem Communications Styles

- ▶ Each **subsystem** provides *services* for other subsystems.
- ▶ There are two different styles of *communication* that make this possible: *client-server* and *peer-to-peer* communication.



The server subsystem does not depend on the client subsystem and is not affected by changes to the client's interface.

Each peer subsystem depends on the other and each is affected by changes in the other's interface.

Subsystem Communications

Client-server communication

- ▶ Client-server communication requires the **client** to know the interface of the **server** subsystem.
 - The communication is only in *one direction*.
- ▶ The **client** subsystem requests *services* from the server subsystem and not vice versa.
 - The **client** plays the role of a *consumer*; while the **server** is considered to be the *supplier*.
- ▶ Examples of client-server communication can be found in most *network-based applications*.
 - E.g. email systems and most web applications.

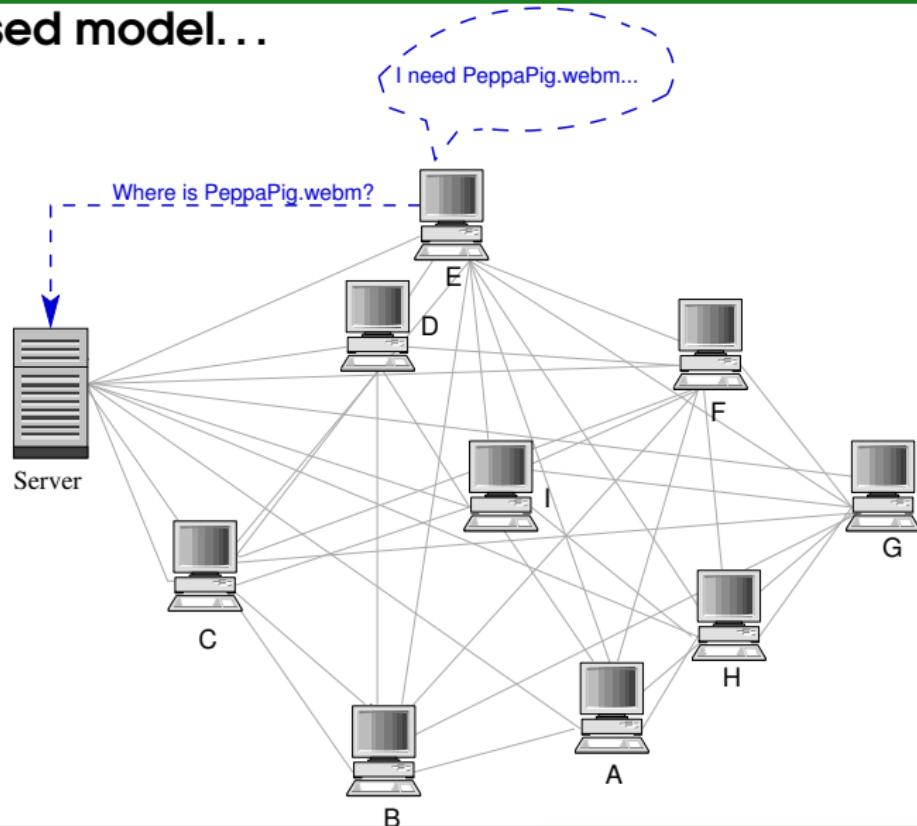
Subsystem Communications

Peer-to-peer communication

- ▶ *Peer-to-peer* (P2P) communication requires each subsystem to know the interface of the other, thus *coupling* them more *tightly*.
- ▶ Each peer has to run exactly the *same program* and hence all peers provide *identical services*.
- ▶ *Peer-to-peer* communication is *two way* since either peer subsystem may request services from the other.

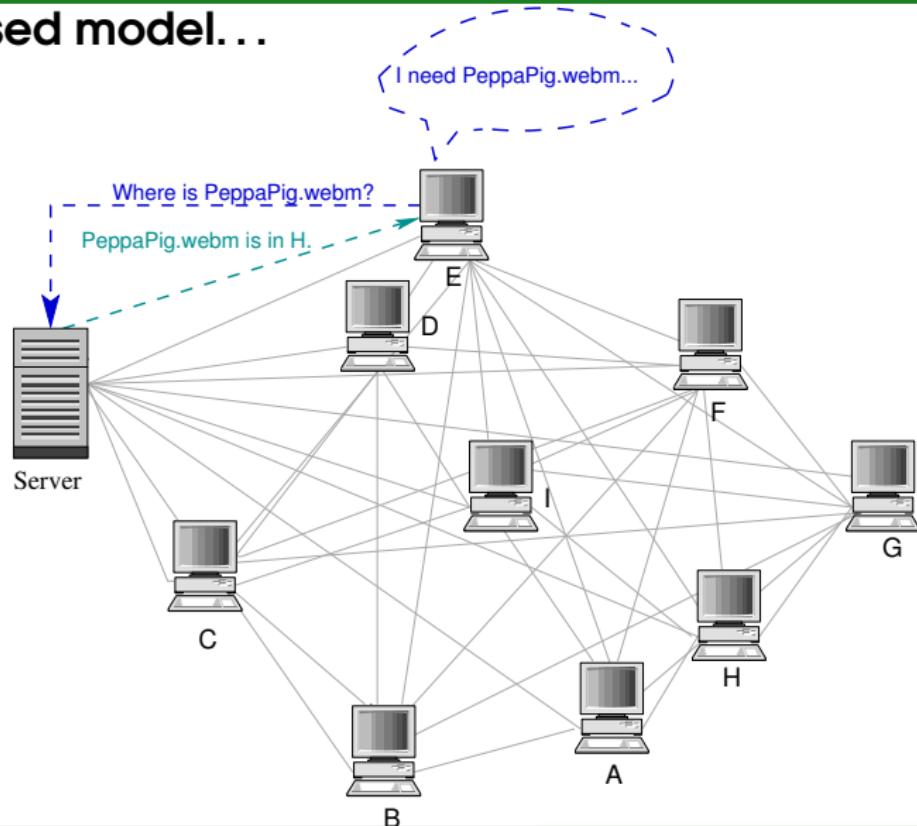
P2P Communication: How to find a peer which holds the required information?

Using a centralised model...



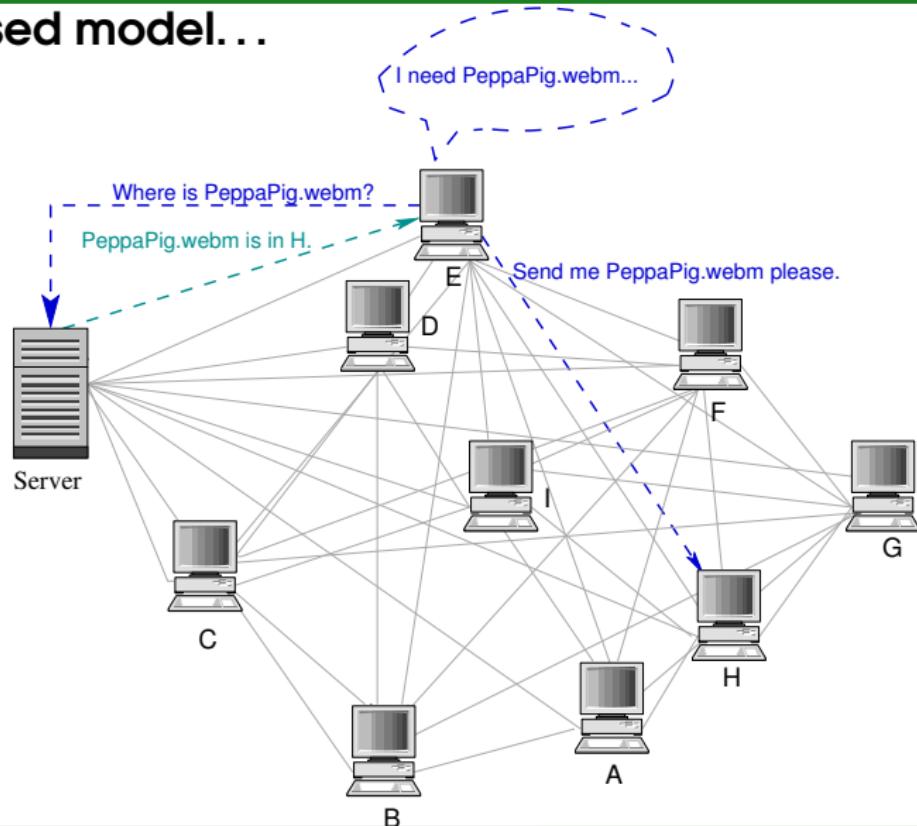
P2P Communication: How to find a peer which holds the required information?

Using a centralised model...



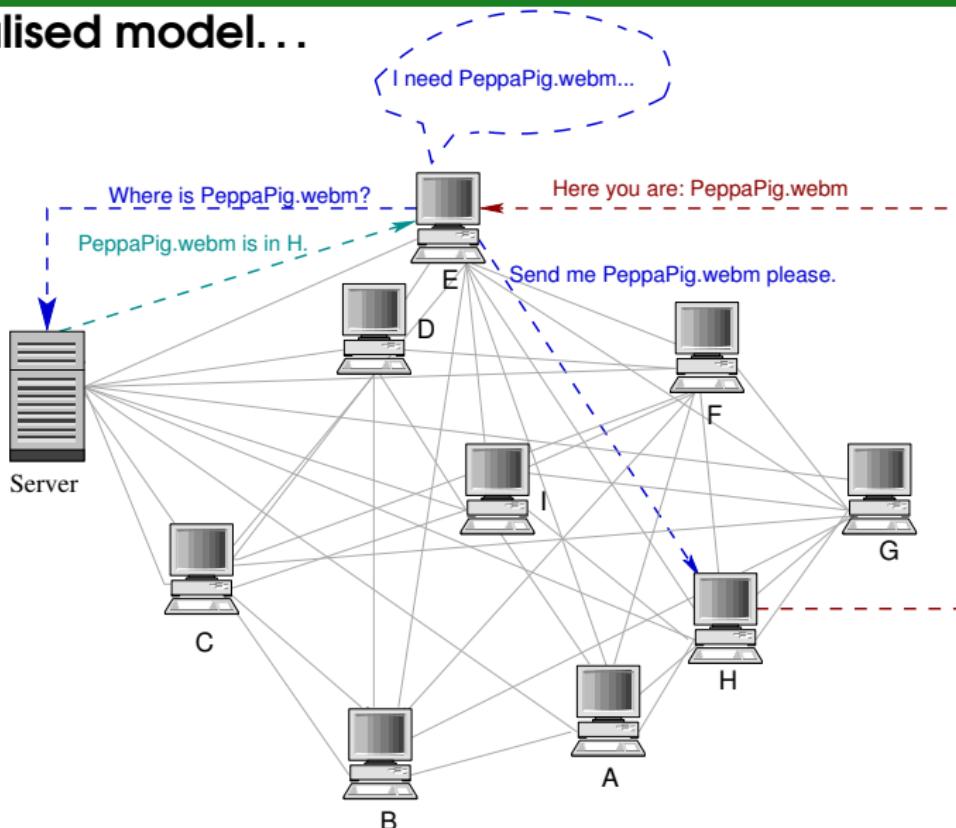
P2P Communication: How to find a peer which holds the required information?

Using a centralised model...



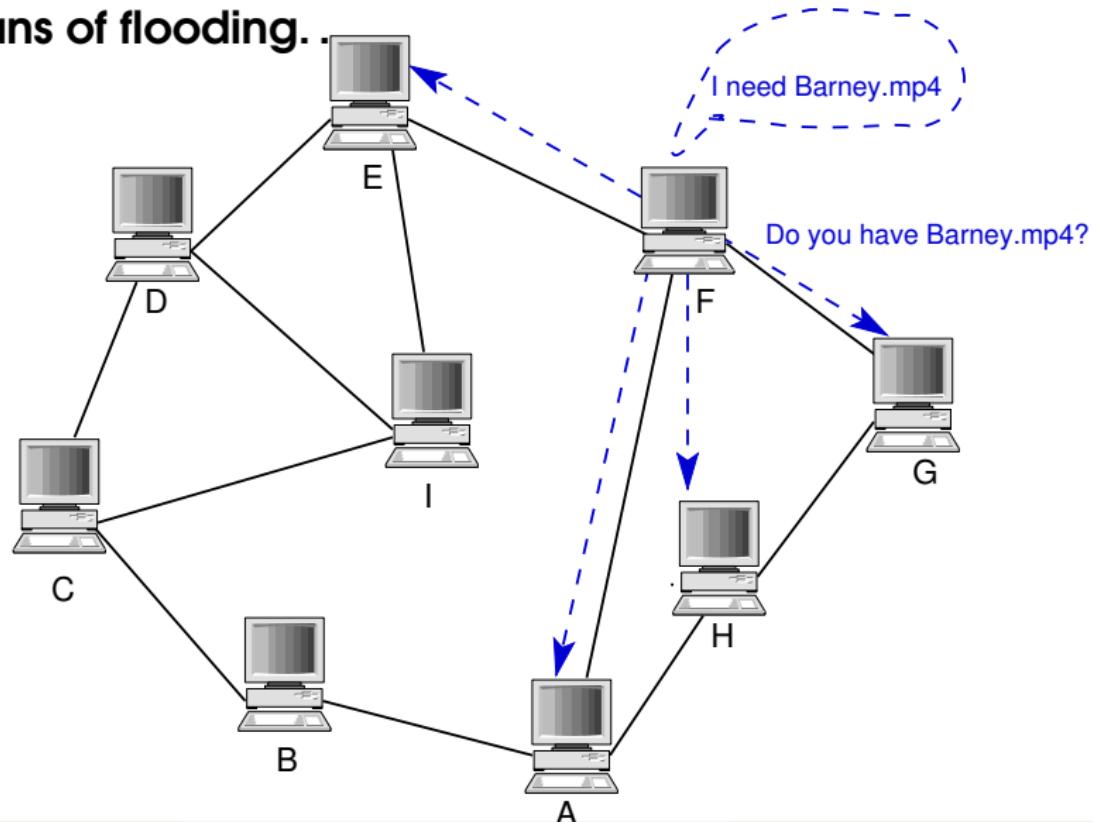
P2P Communication: How to find a peer which holds the required information?

Using a centralised model...



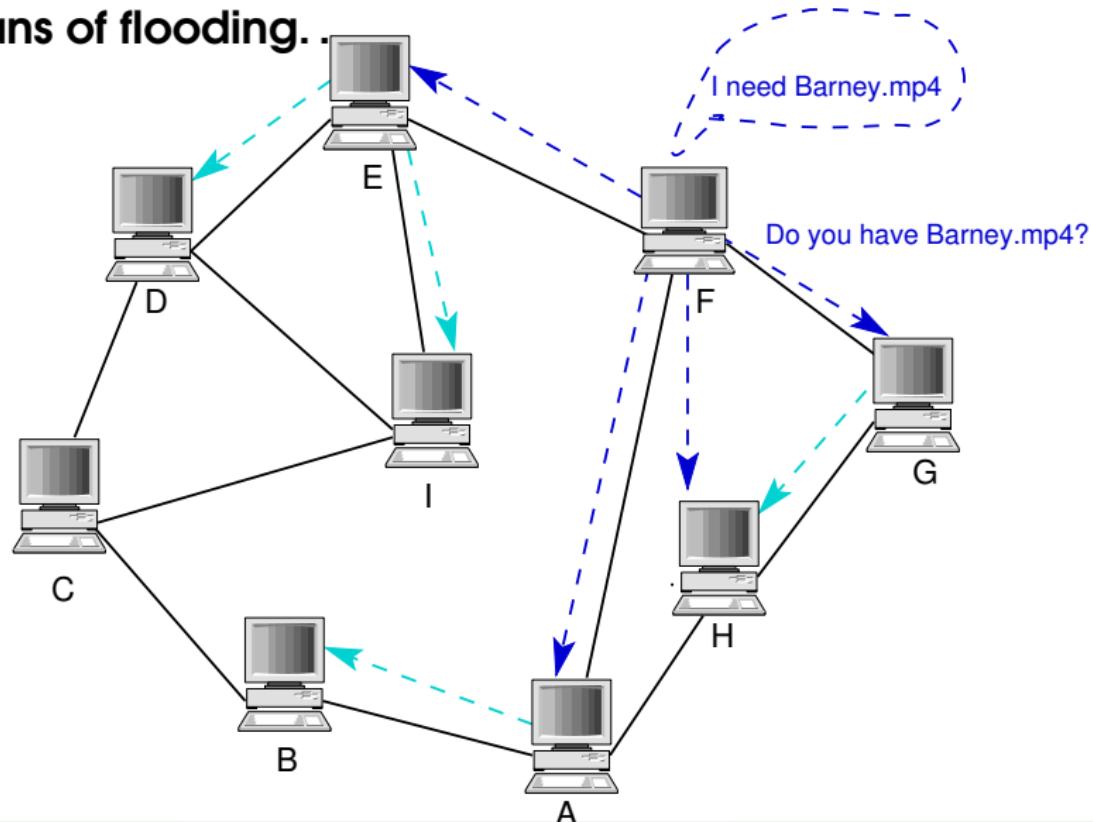
P2P Communication: How to find a peer which holds the required information?

By means of flooding.



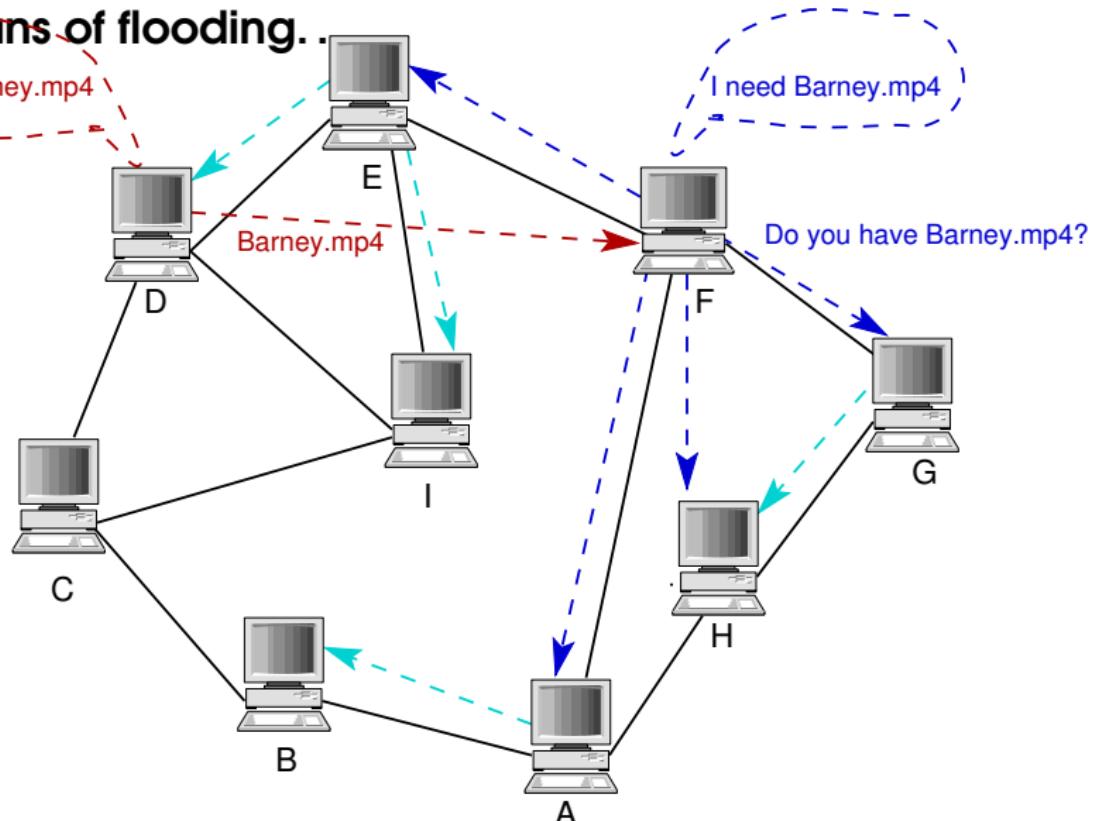
P2P Communication: How to find a peer which holds the required information?

By means of flooding.



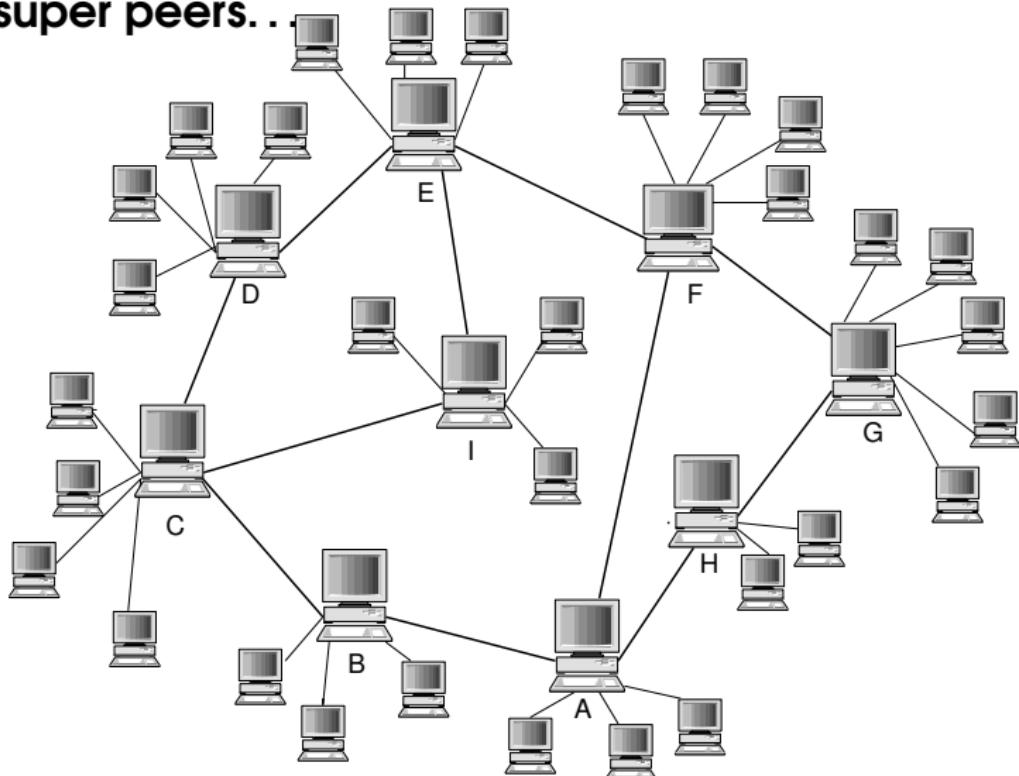
P2P Communication: How to find a peer which holds the required information?

By means of flooding.



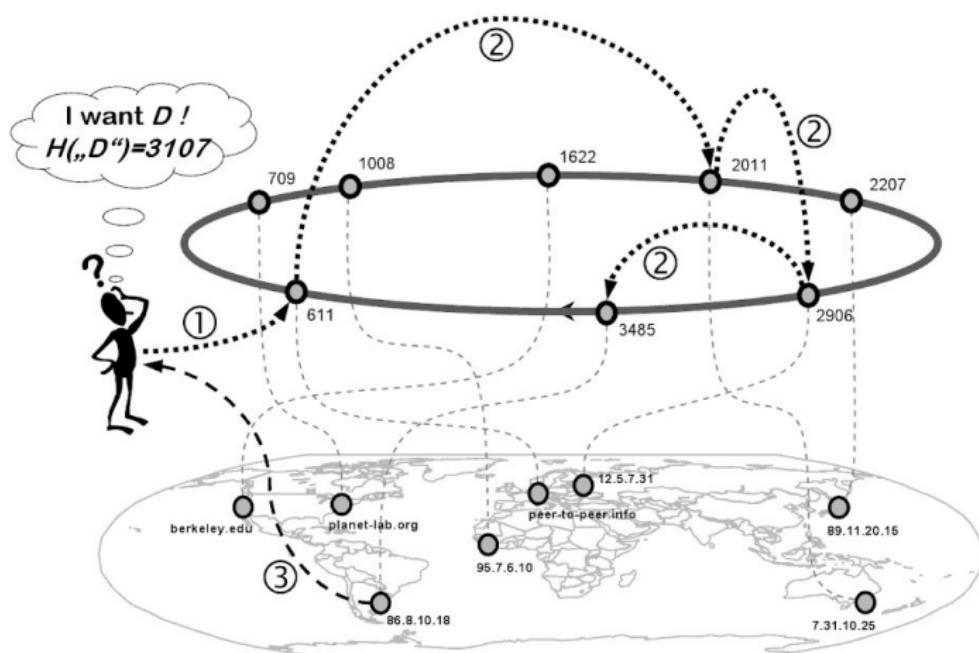
P2P Communication: How to find a peer which holds the required information?

Using ultra-super peers...



P2P Communication: How to find a peer which holds the required information?

Using distributed hash table...



Source: Figure 7.5 (Wehrle, Götz and Rieche, 2005)

Subsystem Communications

Peer-to-peer communication : An example

- ▶ Torrent file sharing systems use a *distributed hash table* to facilitate storing and retrieving of large files.
- ▶ ...*a key-value store distributed across a number of nodes in a network*
 - ⇒ Each peer knows which portions of files it keeps and also where to look for other chunks from other peers.
- ▶ (By contrast, in a blockchain, each node of the network usually stores the full ledger of transactions)
 - ⇒ Example of an attempt to combine approaches: <https://ceur-ws.org/Vol-2334/DLTpaper4.pdf>

Service Oriented Architecture (SOA)

- ▶ Large *distributed* systems may adopt *Service Oriented Architecture (SOA)* in their design.
- ▶ The basic unit of communication in SOA is the invocation of *remote services*.
- ▶ A *service* typically refers to an *Web service*, which:
 - ▶ communicates via Internet protocols (e.g. *HTTP*) and
 - ▶ sends and receives structured data, e.g. in *XML* or *JSON* format.
- ▶ In a SOA, *services* interact via a common communications protocol.
 - ➡ The resulting system components are *loosely coupled* with each other.

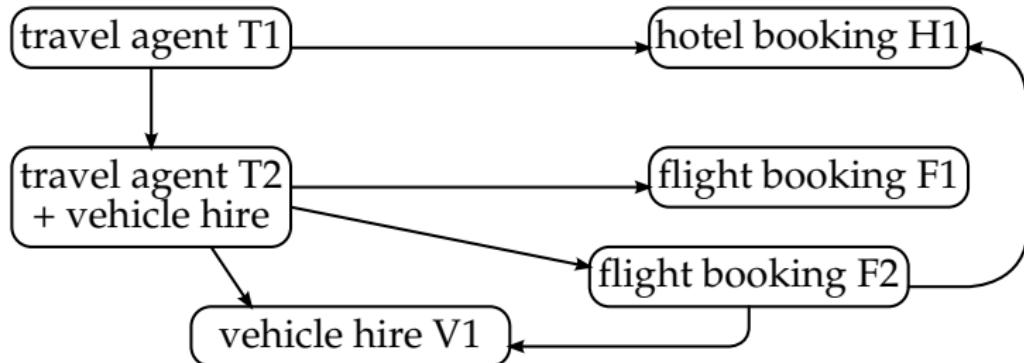
Service Oriented Architecture (SOA)

A Holiday Booking Scenario

- ▶ An example of a SOA system is a network of *Web services* comprising and supporting *travel agents*.
- ▶ This system is composed of *services* for booking flights, hotels and car hire.
- ▶ Agent applications use these *services* to assemble more sophisticated holiday-package services for their clients.

Service Oriented Architecture (SOA)

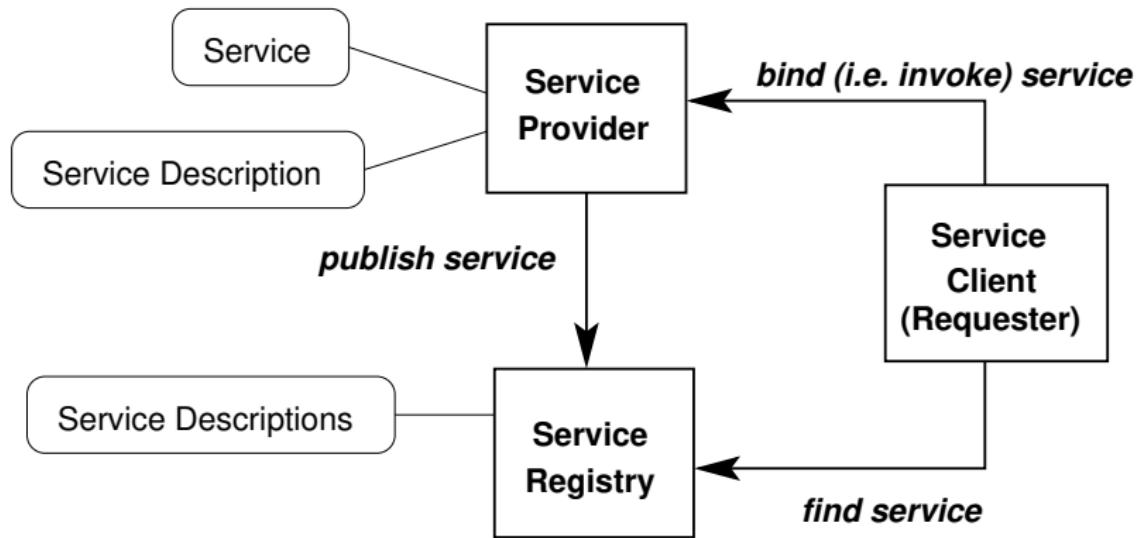
A Holiday Booking Scenario : a SOA example



- ▶ Each bubble is a *service* provided by potentially a *different* vendor.
- ▶ A *service* tends to use another service in order to complete its task.
- ▶ It should be *easy* to switch between the services of the same kind provided by a different vendor.

Service Oriented Architecture (SOA)

Operations in SOA



Papazoglou (2008, Section 1.6.2)

Service Oriented Architecture (SOA)

Common Principles (Erl 2004, Section 3.1.4)

- ▶ Reusable logic is divided into services.
- ▶ Services share a formal contract.
 - mainly regarding information exchange
- ▶ Services are *loosely coupled*.
- ▶ Services *abstract* underlying logic.
 - The only part of a service that is visible to the outside world is what is exposed via the service's description.
- ▶ Services are *composable*.
 - I.e. a service can be made up of other services.

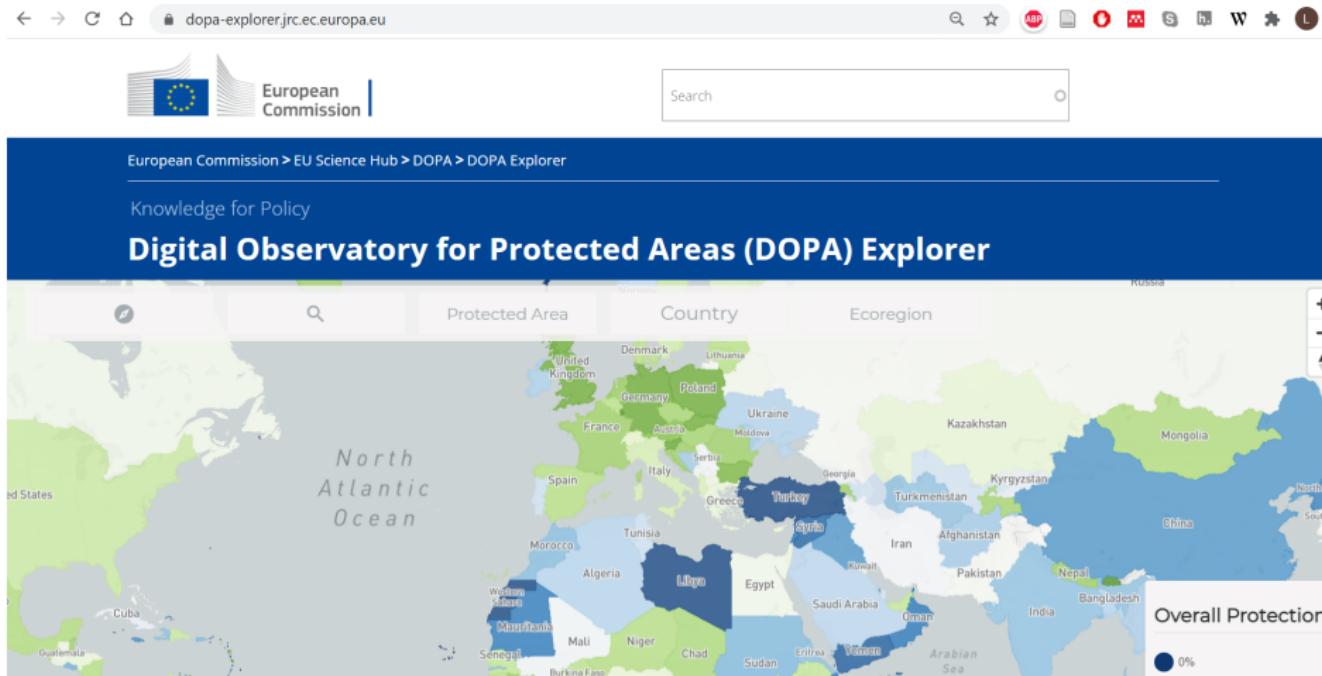
Service Oriented Architecture (SOA)

Why SOA?

Business use of SOA is justified by a promise of:

- ▶ *Easier reuse* of services for multiple purposes;
- ▶ *Better adaptability* to changing business environment and available technologies;
- ▶ Ability to integrate new and *legacy systems*;
- ▶ Ability to *cheaply* setup e-business links across the World.

Web Services - a practical example



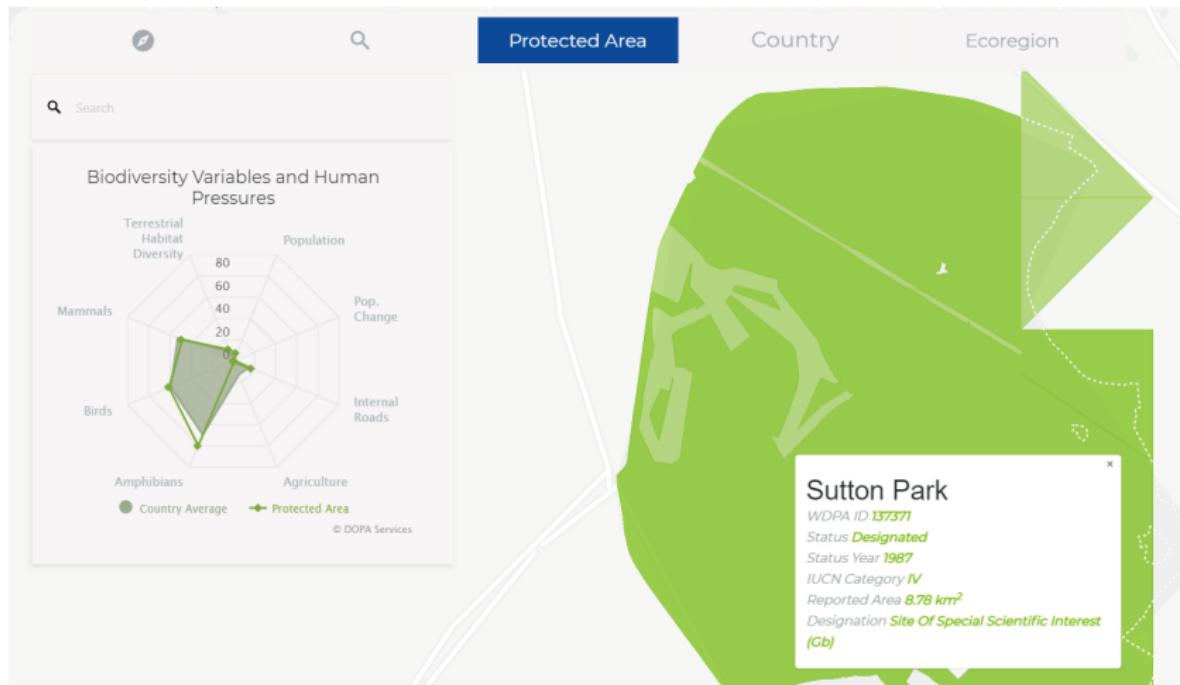
Web Services - a practical example

The screenshot shows a web-based map application titled "Digital Observatory for Protected Areas (DOPA) Explorer". The interface includes a top navigation bar with the European Commission logo and a search bar. Below the navigation, the URL "European Commission > EU Science Hub > DOPA > DOPA Explorer" is visible, along with a "Knowledge for Policy" tagline. The main content area features a map of England and surrounding regions, with green shaded areas representing protected areas. A specific protected area, "Sutton Park", is highlighted with a callout box containing the following details:

Sutton Park
WDPA ID 137371
Status Designated
Status Year 1987
IUCN Category IV
Reported Area 8.78 km²
Designation Site Of Special Scientific Interest (SSSI)

On the left side of the map, there is a sidebar with a search bar, a hexagonal icon, and the text "Calculating statistics for Sutton Park". The map also displays place names like Lincoln, Mansfield, Newark-on-Trent, Nottingham, Leicestershire, Peterborough, and Coventry.

Web Services - a practical example



Web Services - a practical example

← → ⌂ ⌂ 🔒 dopa-explorer.jrc.ec.europa.eu

European Commission |

Digital Observatory for Protected Areas (DOPA) Explorer

Biodiversity Variables and Human Pressures

Terrestrial Habitat Diversity
Population
Internal Roads
Mammals
Birds
Amphibians
Agriculture

Country Average Protected Area

Network

Elements Console Sources Network Performance Memory Application Security

Preserve log Disable cache No throttling

Filter Hide data URLs All XHR JS CSS Img Media Font Doc WS Manifest

Blocked Requests

Use large request rows Group by frame

Show overview Capture screenshots

Name	Status	Type	Initiator	Size
wfs?request=getfeature&version=1.0.0...	200	xhr	jquery.js:9600	1.1 kB
get_de_wdpa_terrestrial_radarplot?form...	200	xhr	jquery.js:9600	2.7 kB
load_gif	200	gif	jquery.js:4773	(disk cache)
168.vector.pbf?access_token=pk.eyJ1...	(canceled)	fetch	96c2011a-0297-4a...	0 B
168.vector.pbf?access_token=pk.eyJ1...	200	fetch	96c2011a-0297-4a...	195 kB
wmts?layer=dopa_explorer_3:dopa_g...	(canceled)	fetch	96c2011a-0297-4a...	0 B
wmts?layer=dopa_explorer_3:global_...	(canceled)	fetch	96c2011a-0297-4a...	0 B
wmts?layer=dopa_explorer_3:global_...	(canceled)	fetch	96c2011a-0297-4a...	0 B
wmts?layer=dopa_explorer_3:global_...	(canceled)	fetch	96c2011a-0297-4a...	0 B
wmts?layer=dopa_explorer_3:dopa_g...	200	fetch	96c2011a-0297-4a...	63.7 kB
wmts?layer=dopa_explorer_3:ecoregi...	(canceled)	fetch	96c2011a-0297-4a...	0 B
wmts?layer=dopa_explorer_3:dopa_g...	200	fetch	96c2011a-0297-4a...	12.8 kB
wmts?layer=dopa_explorer_3:dopa_g...	200	fetch	96c2011a-0297-4a...	0 B
wmts?layer=dopa_explorer_3:global_...	200	fetch	96c2011a-0297-4a...	2.6 kB

129 requests | 593 kB transferred | 1.1 MB resources

Web Services - a practical example

The screenshot shows a web browser window with the URL rest-services.jrc.ec.europa.eu/services/ in the address bar. The page features a large green logo with the letters "DOPA" where the "O" is replaced by a stylized globe. Below the logo is a button labeled "Access Rest Services". The main content area has a white background with a large green border. It contains the heading "Welcome to our web services!" and a paragraph of text. At the bottom, there is a note about terms and conditions.

Welcome to our web services!

You will find here the main documentation of our web services focusing on biodiversity conservation, protected areas, ecosystems, soils, land and marine resources.

These services are documented hereafter using a set of catalogs which have been customized for external users (e.g. Thematic, Datasets, Methods, ...) as well as for internal users (e.g. names of services and databases, responsible developer). This site is continuously updated, has a restricted access and we are currently testing its use by external reviewers from IUCN, UNEP-WCMC and Birdlife International.

Any feedback is welcome!

Use of these web services is subject to the terms and conditions of use governing the original datasets. These restrict the use of the data for commercial purposes and require users to acknowledge the source of the data. JRC would like to be informed should the web services be used regularly.
For full terms and conditions see [here](#).

Web Services - a practical example

The screenshot shows a web browser window for the 'JOINT RESEARCH CENTRE REST SERVICES DIRECTORY'. The URL is rest-services.jrc.ec.europa.eu/services/services/?thematic_name=species. The page header includes links for Print, Accessibility help, A to Z, Sitemap, About this site, Privacy Statement, Legal notice, Cookies, Contact, and Search, along with a dropdown menu for selecting a value. The main content area features the European Commission logo and the DOPA logo. The navigation bar shows the path: European Commission > EU Science Hub > DOPA > Services. Below the navigation is a toolbar with icons for Home, Settings, Help, and a user ID n0036026. The main content area displays 'Rest Services Categories' and a 'Full list of available categories'. It includes filters for Thematics, INSPIRE, Coverage, Reporting Level, Datasets, and eServices. An applied filter for 'THEMATICS: species' is shown, with a description: 'DESCRIPTION: Information related to species distribution using information from IUCN Red List of Threatened Species and other datasets'.

Web Services - a practical example

Show 10 entries

Search:

#.ID	Function Name	Function Description	Edit
4571	get_list_species_output	Shows all species direct and relate attributes.	
4575	get_single_species_output	Shows for a single species direct and related detailed attributes	
4712	get_dopa_country_iucn_threatened_endemic_vertebrates	.	
4713	get_dopa_country_iucn_species	.	
4714	get_dopa_country_iucn_threatened_animals	.	
4720	get_dopa_wdpa_redlist_list	.	
4721	get_dopa_wdpa_redlist_status	.	
4722	get_dopa_country_pa_redlist_count	.	
4723	get_dopa_country_pa_redlist_indicator	.	
4728	get_dopa_country_pa_redlist_list	.	

Showing 1 to 10 of 10 entries

Previous 1 Next

Web Services - a practical example

← → ⌂ ⌂ 🔒 dopa-explorer.jrc.ec.europa.eu

European Commission

Digital Observatory for Protected Areas (DOPA) Explorer

Biodiversity Variables and Human Pressures

Population
Internal Roads
Agriculture
Amphibians
Birds
Mammals
Terrestrial Habitat Diversity

Protected Area
Country Average

Network

Filter

Preserve log
Disable cache
No throttling

Blocked Requests

Use large request rows
Show overview

Group by frame
Capture screenshots

Name	Headers	Preview	Response	Initiator
wfs?request=getfeature&version=1.0.0&service=.				
get_de_wdpa_terrestrial_radarplot?format=json&			<pre>1 { 2 "records": [3 { 4 "output_order": 1, 5 "indicator": "ppi", 6 "title": "Population", 7 "country_avg": 1.95, 8 "site_norm_value": 7.34 9 }, 10 { 11 "output_order": 2, 12 "indicator": "ppi_change", 13 "title": "Popn. change", 14 "country_avg": 1.39, 15 "site_norm_value": 0.91 16 }, 17 { 18 "output_order": 3, 19 "indicator": "rp_in", 20 "title": "Internal Roads". 21 } 22] 23}</pre>	

129 requests | 593 kB transferred | 1.1 MB resources | Line 1, Column 1

Web Services - a practical example

#.ID	Function Name	Function Description
4571	get_list_species_output	Shows all species attributes.
4575	get_single_species_output	Shows for a single related detailed attributes.
4712	get_dopa_country_iucn_threatened_endemic_vertebrates	.
4713	get_dopa_country_iucn_species	.
4714	get_dopa_country_iucn_threatened_animals	.
4720	get_dopa_wdpa_redlist_list	.
4721	get_dopa_wdpa_redlist_status	.

Web Services - a practical example

rest-services.jrc.ec.europa.eu/services/d6dopa/species/get_single_species_output

This service uses information from the IUCN Red List of Threatened Species™. Use of this web service is subject to the terms and conditions of use governing the original dataset. These restrict the use of the data for commercial purposes and require users to acknowledge the source of the data. The web service does not provide access to the original reference datasets available from their respective sources – users wishing to download the original data are invited to consult the authoritative sources as follows, where the relevant terms and conditions of use are also available: <https://www.iucnredlist.org/resources/spatial-data-download>

This service uses distribution data (digitized maps) compiled by BirdLife International. Use of this web service is subject to the terms and conditions of use governing the original dataset. These restrict the use of the data for commercial purposes and require users to acknowledge the source of the data. The web service does not provide access to the original reference datasets available from their respective sources – users wishing to download the original data are invited to consult the authoritative sources as follows, where the relevant terms and conditions of use are also available: <http://datazone.birdlife.org/species/requestdis>

Input parameters:

Name	Description	Type	Compulsory	Default value
a_id_no	filters by (single) species id	bigint		3746

Web Services - a practical example

REST call using optional parameters
?a_id_no=3746&format=xml

Excel endpoint
?format=xml

Create your REST call, select parameters and format, submit to get the query string.

Optional parameters	Format	Options
<input type="checkbox"/> a_id_no Value: 3746	- Select a Value - <input type="checkbox"/> includemetadata <input type="checkbox"/> decimals <input type="checkbox"/> 2 <input type="checkbox"/> sortfield - Select a Value -	<input type="checkbox"/> fields <input type="checkbox"/> id_no <input type="checkbox"/> class <input type="checkbox"/> order_ <input type="checkbox"/> family ...

Web Services - a practical example

rest-services.jrc.ec.europa.eu/services/d6dopa/species/get_single_species_output?a_id_no=3746&format=xml

```
<results>
  <records>
    <record binomial="Canis lupus" category="LC" class="MAMMALIA" conservation_needs="1.1 - Site/area protection; 1.2 - Resource & habitat management; 2.3 - Habitat & natural process restoration; 3.1.1 - Harvest management; 3.1.3 - Limiting population growth; 3.2 - Species awareness & communications; 5.1.1 - International level; 5.1.2 - National level; 5.2 - Policies and regulations; 5.4.1 - International level; 5.4.3 - Sub-national level; 6.4 - Conservation payments" countries="AE - United Arab Emirates; AF - Afghanistan; AL - Albania; AM - Armenia; BA - Bosnia and Herzegovina; BE - Belgium; BG - Bulgaria; BT - Bhutan; BY - Belarus; CA - Canada; CH - Switzerland; CN - China; CZ - Czechia; DE - Germany; DK - Djibouti; EE - Estonia; ES - Spain; FI - Finland; FR - France; GE - Georgia; GL - Greenland; GR - Greece; HR - Croatia; HU - Hungary; IL - Israel; IS - Iceland; IT - Italy; JO - Jordan; KG - Kyrgyzstan; KP - Korea, Democratic People's Republic of; KR - Korea, Republic of; LK - Sri Lanka; LV - Latvia; LY - Libya; MD - Moldova; ME - Montenegro; MK - North Macedonia; MM - Myanmar; MN - Mongolia; MX - Mexico; NE - Niger; NL - Netherlands; OM - Oman; PK - Pakistan; PL - Poland; PT - Portugal; RO - Romania; RS - Serbia; RU - Russian Federation; SA - Saudi Arabia; SK - Slovakia; SY - Syrian Arab Republic; TJ - Tajikistan; TM - Turkmenistan; TR - Turkey; UA - Ukraine; US - United States; UZ - Uzbekistan" ecosystems="terrestrial" endemic="False" family="CANIDAE" genus="Canis" habitats="1.1 - Forest - Boreal; 1.2 - Forest - Subarctic; 1.3 - Forest - Temperate; 1.4 - Forest - Subtropical/Tropical Dry; 1.5 - Forest - Subtropical/Wet; 1.6 - Forest - Tropical Rainforest; 1.7 - Forest - Tropical Wetlands (incl. pools and temporary waters from snowmelt); 1.8 - Forest - Alpine; 2.1 - Shrubland - Subarctic; 2.2 - Shrubland - Temperate; 2.3 - Shrubland - Mediterranean-type Shrubby Vegetation; 2.4 - Grassland - Tundra; 2.5 - Grassland - Subarctic; 2.6 - Grassland - Temperate; 2.7 - Grassland - Alpine; 2.8 - Grassland - Páramo; 2.9 - Grassland - Savanna; 2.10 - Grassland - Mangrove; 3.1 - Shrubland - Subarctic; 3.2 - Shrubland - Temperate; 3.3 - Shrubland - Mediterranean-type Shrubby Vegetation; 3.4 - Shrubland - Subtropical/Wet; 3.5 - Shrubland - Tropical Rainforest; 3.6 - Shrubland - Alpine; 3.7 - Shrubland - Mangrove; 4.1 - Grassland - Tundra; 4.2 - Grassland - Subarctic; 4.3 - Grassland - Temperate; 4.4 - Grassland - Alpine; 4.5 - Grassland - Mangrove; 4.6 - Grassland - Savanna; 4.7 - Grassland - Páramo; 4.8 - Grassland - Subtropical/Wet; 4.9 - Grassland - Subtropical/Tropical Dry; 4.10 - Grassland - Tropical Rainforest; 5.1 - Intentional use (species is the target); 5.2 - Persecution/control; 5.3 - Recreational activities" usetrade="10 - Wearing hunting/specimen collecting"/>
  </records>
</results>
```

Web Services - a practical example

rest-services.jrc.ec.europa.eu/services/d6dopa/species/get_single_species_output?a_id_no=3746&format=json

```
{  
  "records": [  
    {  
      "id_no": 3746,  
      "class": "MAMMALIA",  
      "order": "CARNIVORAT",  
      "family": "CANIDAE",  
      "genus": "Canis",  
      "binomial": "Canis lupus",  
      "category": "LC",  
      "threatened": false,  
      "n_country": 68,  
      "endemic": false,  
      "ecosystems": "terrestrial",  
      "habitats": "1.1 - Forest - Boreal; 1.2 - Forest - Subarctic; 14.1 - Artificial/Terrestrial - Arable Land; 14.2 - Artificial/Terrestrial - Pastureland; 14.3 - Artificial/Terrestrial - Plantations; 14.4 - Artificial/Terrestrial - Rural Gardens; 1.4 - Forest - Temperate; 1.5 - Forest - Subtropical/Tropical Dry; 3.1 - Shrubland - Subarctic; 3.3 - Shrubland - Boreal; 3.4 - Shrubland - Temperate; 3.5 - Shrubland - Subtropical/Tropical Dry; 3.8 - Shrubland - Mediterranean-type Shrubby Vegetation; 4.1 - Grassland - Tundra; 4.2 - Grassland - Subarctic; 4.4 - Grassland - Temperate; 5.10 - Wetlands (inland) - Tundra Wetlands (incl. pools and temporary waters from snowmelt); 5.11 - Wetlands (inland) - Alpine Wetlands (includes temporary waters from snowmelt); 5.4 - Wetlands (inland) - Bogs, Marshes, Swamps, Fens, Peatlands; 6 - Rocky areas (eg. inland cliffs, mountain peaks); 8.1 - Desert - Hot; 8.2 - Desert - Temperate; 8.3 - Desert - Cold",  
      "countries": "AE - United Arab Emirates; AF - Afghanistan; AL - Albania; AM - Armenia; AT - Austria; AZ - Azerbaijan; BA - Bosnia and Herzegovina; BE - Belgium; BG - Bulgaria; BT - Bhutan; CA - Canada; CH - Switzerland; CN - China; CZ - Czechia; DE - Germany; DK - Denmark; EE - Estonia; ES - Spain; FI - Finland; FR - France; GE - Georgia; GL - Greenland; GR - Greece; HR - Croatia; HU - Hungary; IL - Israel; IN - India; IQ - Iraq; IR - Iran, Islamic Republic of; IT - Italy; JO - Jordan; KG - Kyrgyzstan; KP - Korea, Democratic People's Republic of; KR - Korea, Republic of; KZ - Kazakhstan; LT - Lithuania; LU - Luxembourg; LV - Latvia; LY - Libya; MD - Moldova; ME - Montenegro; MK - North Macedonia; MM - Myanmar; MN - Mongolia; MX - Mexico; NL - Netherlands; NO - Norway; NP - Nepal; OM - Oman; PK - Pakistan; PL - Poland; PT - Portugal; RO - Romania; RS - Serbia; RU - Russian Federation; SA - Saudi Arabia; SE - Sweden; SI - Slovenia; SK - Slovakia; SY - Syrian Arab Republic; TJ - Tajikistan; TM - Turkmenistan; TR - Turkey; UA - Ukraine; US - United States; UZ - Uzbekistan; YE - Yemen",  
      "stresses": "1.1 - Ecosystem conversion; 1.2 - Ecosystem degradation; 2.1 - Species mortality; 2.2 - Species disturbance",  
      "threats": "11.1 - Habitat shifting & alteration; 1.1 - Housing & urban areas; 1.2 - Commercial & industrial areas; 1.3 - Tourism & recreation areas; 2.3.1 - Nomadic grazing; 2.3.2 - Small-holder grazing, ranching or farming; 2.3.3 - Agro-industry grazing, ranching or farming; 4.1 - Roads & railroads; 5.1.1 - Intentional use (species is the target); 5.1.3 - Persecution/control; 6.1 - Recreational activities",  
      "research_needs": "1.1 - Taxonomy; 1.2 - Population size, distribution & trends; 1.3 - Life history & ecology; 1.4 - Harvest, use & livelihoods; 1.5 - Threats; 1.6 - Actions; 2.1 - Species Action/Recovery Plan; 2.2 - Area-based Management Plan; 2.3 - Harvest & Trade Management Plan; 3.1 - Population trends; 3.2 - Harvest level trends; 3 - Trade trends; 3.4 - Habitat trends",  
      "conservation_needs": "1.1 - Site/area protection; 1.2 - Resource & habitat protection; 2.1 - Site/area management; 2.3 - Habitat & natural process restoration; 3.1.1 - Harvest management; 3.1.3 - Limiting population growth; 3.2 - Species recovery; 4.1 - Formal education; 4.3 - Awareness & communications; 5.1.1 - International level; 5.1.2 - National level; 5.2 - Policies and regulations; 5.4.1 - International level; 5.4.2 - National level; 5.4.3 - Sub-national level; 6.4 - Conservation payments",  
      "use/trade": "10 - Wearing apparel, accessories; 15 - Sport hunting/specimen collecting"  
    }  
  ]  
}
```

Web Services - a practical example



European Commission > EU Science Hub > DOPA > DOPA Explorer > Sutton Park

Knowledge for Policy

Digital Observatory for Protected Areas (DOPA) Explorer

Species list for protected area

The following species list is computed from the species ranges recorded in the IUCN Red List of Threatened Species.

Show 10 entries

Search

IUCN ID	Scientific Name	Phylum	Class	Order	Family	Status
	<i>Bufo bufo</i>	Chordata	Amphibia	Anura	Bufonidae	Least Concern (LC)
	<i>Lissotriton</i>	Chordata	Amphibia	Caudata	Salamandridae	Least Concern (LC)

A map showing the location of Sutton Park, a large green area outlined in orange, within a satellite view of the surrounding urban and rural landscape. The map includes labels for "Four Oaks", "Streatly", "Kingstanding", and "Sutton". A legend on the left side of the map shows icons for zooming, navigating, and saving the map.

References

- ▶ Hawa, M., "Cooperation Incentives: Issues and Design Strategies", in Antonopoulos, N., Exarchakos, G., Li, M. and Liotta, A. (eds), *Handbook of Research on P2P and Grid Systems for Service-Oriented Computing: Models, Methodologies, and Applications*, Volume 1, New York: Information Science Reference, 2010.
- ▶ Arkin, A., Askary, S., Fordin, S., Jekeli, W., Kawaguchi, K., Orchard, D., Pogliani, S., Riemer, K., Struble, S., Takacs-Nagy, P., Trickovic, I. and Zimek, S., *Web Service Choreography Interface (WSCI) 1.0*, W3C Note, 8 August 2002, (Online). Available at: <http://www.w3.org/TR/wsci/> (17/10/2019).

References

- ▶ Bennett, S., McRobb, S. and Farmer, R., *Object-Oriented Systems Analysis and Design Using UML*, 4th ed, Maidenhead: McGraw-Hill, 2010.
- ▶ Booch, G., Rumbaugh, J. and Jacobson, I., *The UML specification documents*, Santa Clara, CA.: Rational Software Corp., 1997.
- ▶ Braude, E.J. and Bernstein, M.E., *Software Engineering: Modern Approaches*, 2nd ed, Hoboken, NJ: Wiley, 2011.
- ▶ Eckstein, R., *Java SE Application Design With MVC*, 2007, (Online). Available at: <http://www.oracle.com/technetwork/articles/javase/index-142890.html> (17/10/2019).

References

- ▶ Eeles, P., *What is a software architecture?*, 2006, (Online). Available at: <http://www.ibm.com/developerworks/rational/library/feb06/eeles/> (17/10/2019).
- ▶ Erl, T., *Service-Oriented Architecture: A field guide to integrating XML and Web Services*, Upper Saddle River, New Jersey: Pearson Education, 2004.
- ▶ Haas, H. and Brown, A., *Web Services Glossary*, W3C Working Group Note, 11 February 2004, (Online). Available at: <http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/> (17/10/2019).
- ▶ Jacobson, I., Booch, G. and Rumbaugh, J., *The Unified Software Development Process*, Reading, MA: Addison-Wesley; ACM Press, 1999.

References

- ▶ IEEE Computer Society, IEEE Recommended Practice for Architectural Description of Software-Intensive Systems: IEEE Std 1472000, 2000.
- ▶ Kruchten, P., *The Rational Unified Process: An Introduction*, Reading, MA: Addison-Wesley, 2004.
- ▶ Larman, C., *Applying UML and patterns*, 3rd ed, Upper Saddle River, NJ: Prentice Hall, 2005.
- ▶ Lunn, K., *Software Development with UML*, London: Palgrave Macmillan, 2003.

References

- ▶ Microsoft MSDN Library, *Physical Tiers and Deployment*, chapter 19, 2011, (Online). Available at: <http://msdn.microsoft.com/en-us/library/ee658120.aspx> (17/10/2019).
- ▶ Papazoglou, M., *Web Services: Principles and Technology*, Harlow: Pearson Education, 2008.
- ▶ Ramirez, A.O., 'Three-Tier Architecture', *Linux Journal*, Issue 75, July, 2000.
- ▶ Reenskaug, T.M.H., *Model-View-Controller (MVC)*, (Online). Available at: <http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html> (17/10/2019).

References

- ▶ Wehrle, K., Götz, S. and Rieche, S., "Distributed Hash Table", in Steinmetz, R. and Wehrle, K. (ed), *Peer-to-peer Systems and Applications*, LNCS 3485, Berlin: Springer-Verlag, 2005, pp. 79–93.
- ▶ Shaw, M. and Garlan,D., *Software Architecture – Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
- ▶ Swartz, F., *Model-View-Controller (MVC) Structure*, 2004, (Online). Available at:
<https://wwwleepoint.net/GUI/structure/40mvc.html>
(17/10/2019).

For more detail, see:

Chapter 13 of Bennett et al. (2010) and
Chapter 18 of Braude & Bernstein (2011).

Learning Outcomes

Learning Outcomes. You should now be able to

- ▶ explain what is meant by software architecture
- ▶ specify the role of a system architect
- ▶ describe the characteristics of a range of architectural styles
- ▶ identify if a simple software system follows the MVC architecture
- ▶ apply the MVC architecture

Unit 9 Object-Oriented Detailed Design

Unit Outcomes. Here you will learn

- ▶ how to represent **classes** and their relationships in UML class diagrams
- ▶ how to design **associations**
- ▶ how to evaluate a detailed design in terms of **cohesion** and **coupling**
- ▶ what is meant by the **Liskov Substitution Principle (LSP)** and how it helps to ensure a good design

Further Reading: Bennett et al. (2010) Ch14
and Braude & Bernstein (2011) Ch16

based on Bennett, McRobb & Farmer (2010), Chapter 15

Detailed Design: Classes

Introduction

- ▶ In system analysis, key business concepts are identified and expressed as *classes*, which capture key business concepts and operations, but only present a *partial* view of the required model.
- ▶ In detailed design, more detail is added to the class model, which includes:
 - ▶ Deciding the data type of each *attribute*.
 - ▶ Deciding how to handle derived attributes.
 - ▶ Adding primary *operations*.
 - ▶ Defining the *signature* of operations including the types of parameters.
 - ▶ Deciding the *visibility* of attributes and operations.

Style Guidelines for UML Class Diagrams

OMG (2010, p.52) specifies guidelines for class diagrams:

- ▶ *Class* name should be *centered* and presented in *bold*.
- ▶ The first letter of class names should be *capitalized* (if the character set supports uppercase).
- ▶ *Attributes* and *operations* names should begin with a *lowercase* letter.
- ▶ The name of each *abstract class* should appear in *italics*.
- ▶ *Full* attributes and operations should be shown *only* when needed. They should be *suppressed* in other contexts or when merely referring to a class.

UML Class Diagrams: Basic Notations

Specifying Attributes : Examples

Show a derived attribute
balance plus overdraft.

lower case

BankAccount	
nextAccountNumber:	Integer
accountNumber:	Integer
accountName:	String {not null}
balance:	Money = 0.00
/availableBalance:	Money
overdraftLimit:	Money
<hr/>	
open(accountName: String):	Boolean
close():	Boolean
credit(amount: Money):	Boolean
debit(amount: Money):	Boolean
viewBalance():	Money
getBalance():	Money
setBalance(newBalance: Money)	
getAccountName():	String
setAccountName(newName: String)	

Class name centres, bold Capital first letter.

BankAccount class with the attribute data types included

only showing much detail if relevant.

Specifying Attributes : Examples

- The attribute `balance` in class `BankAccount` might be declared with an *initial value* of zero using the syntax:

```
balance:Money = 0.00
```

- Attributes that may not be `null` are specified using a *property string*:

```
accountName:String {not null}
```

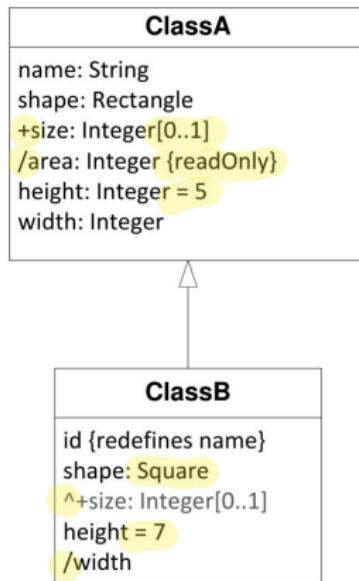
- Arrays are specified using *multiplicity*:

```
qualifications:String[0..10]
```

- The *derived attribute* `availableBalance` whose value is calculated from `balance + overdraftLimit` is marked by `/`:

```
/availableBalance: Money
```

Specifying Attributes : More Examples (OMG 2013, p.115)



- ▶ ClassA::name is an attribute with type String.
- ▶ ClassA::shape is an attribute with type Rectangle.

- ▶ ClassA::size is a *public* attribute of type Integer with *multiplicity* 0..1.
- ▶ ClassA::area is a *derived attribute* with type Integer and a *read-only* property.
- ▶ ClassA::height is an attribute of type Integer with a *default initial value* of 5.
- ▶ ClassA::width is an attribute of type Integer.
- ▶ ClassB::shape is an attribute that *redefines* ClassA::shape. It has type Square, a specialization of Rectangle.
- ▶ ClassB shows size as an attribute *inherited* from ClassA, as signified by the prepended *caret* symbol.
- ▶ ClassB::height is an attribute that *redefines* ClassA::height. It has a *default* value of 7 for ClassB instances that overrides the ClassA *default* of 5.
- ▶ ClassB::width is a *derived attribute* that *redefines* ClassA::width, which is not derived.

- ▶ An operation's *signature* is determined by the operation's *name*, the *number* and the *type of its parameters* and the *return type*, if any.
- ▶ *visibility* is the visibility of the operation, i.e. one of the following:
 - I.e. +public, -private, #protected, ~package
- ▶ *name* is the name of the operation
 - The *name* of the operation followed by a pair of brackets is the *mandatory* part of specifying an operation.
- ▶ The *parameter-list* is *optional*. If included, it is a list of *parameter names and types* separated by commas.
- ▶ *return-type* is the type of the return value if the operation has one defined

Operation Signatures : Examples

- ▶ BankAccount might have a credit() operation that would be shown in the diagram as:

```
credit(amount: Money) : Boolean
```

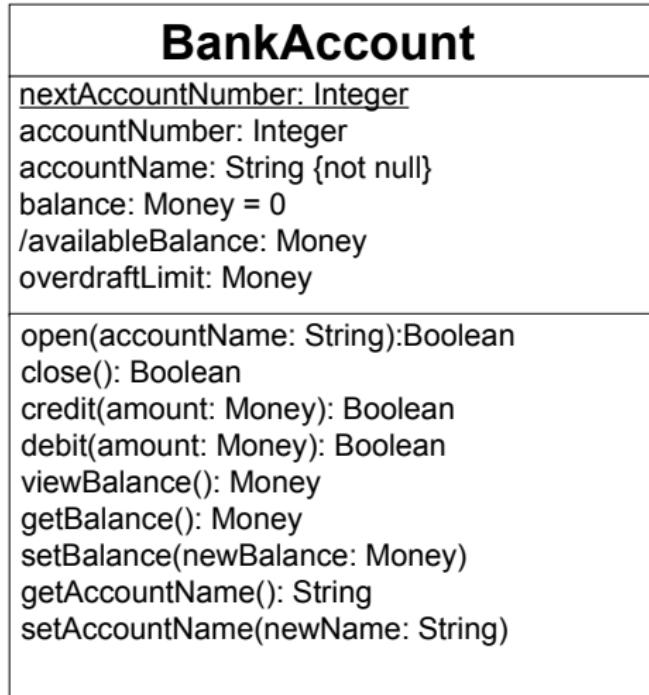
parameter type *return type*

- ▶ More examples from OMG (2013, p.120):

```
- hide()  
  
+ toString() : String  
  
+ insertionSort<T -> Comparable>(data : T[1..*])
```

UML Class Diagrams: Basic Notations

Operation Signatures : Examples



BankAccount
class with
operation
signatures
included.

UML Class Diagrams: Basic Notations

Which operations to include?

Here are some guidelines for determining which operations to be shown in a class diagram:

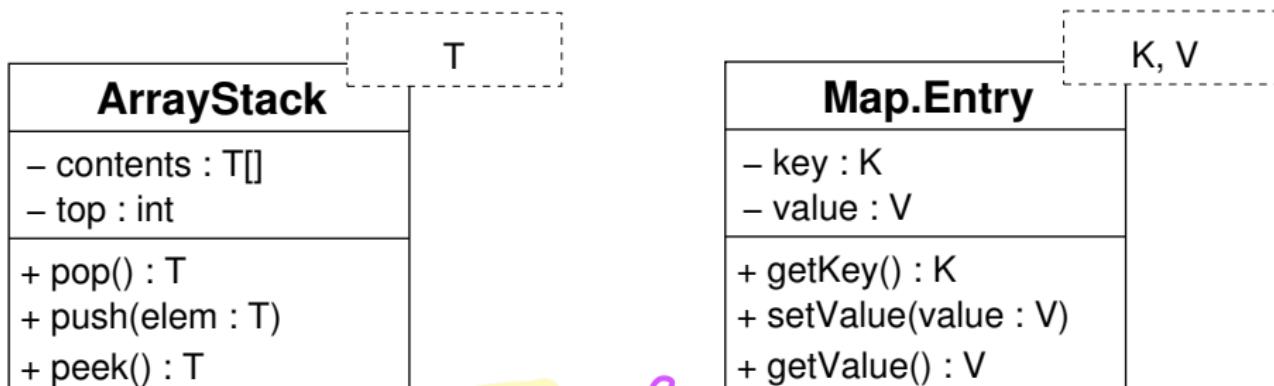
- ▶ Show *primary operations* (i.e. *constructors*, *destructors*, *getters* and *setters*) where it is *necessary*.
- ▶ Only show constructors where they have *special significance*.
- ▶ *Varying levels of detail* at different stages in the development cycle.

UML Class Diagrams: Basic Notations

Template : Examples

Templates are model Elements that are parameterized by other model Elements. (OMG 2013, p.15)

Templates typically appear within the context of specifying a generic type.



The variable type of T, K and V is not specified at this point.

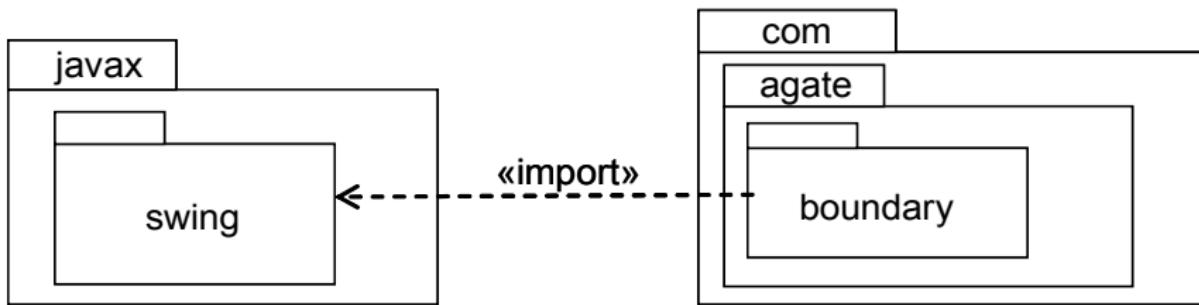
Class Relations

OMG (2010, p.144) specifies notations for various types of class relations:

PATH TYPE	NOTATION	REFERENCE
Aggregation		See 7.3.2, 'AggregationKind (from Kernel)'
Association		See 7.3.3, 'Association (from Kernel)'
Composition		See 7.3.2, 'AggregationKind (from Kernel)'
Dependency		See 7.3.12, 'Dependency (from Dependencies)'
Generalization		See 7.3.20, 'Generalization (from Kernel, PowerTypes)'
InterfaceRealization		See 7.3.25, 'InterfaceRealization (from Interfaces)'
Realization		See 7.3.45, 'Realization (from Dependencies)'
Usage	 	See 7.3.53, 'Usage (from Dependencies)' e.g. word describes the usage (import)

Class Relations

The following diagram shows an *import* relationship:

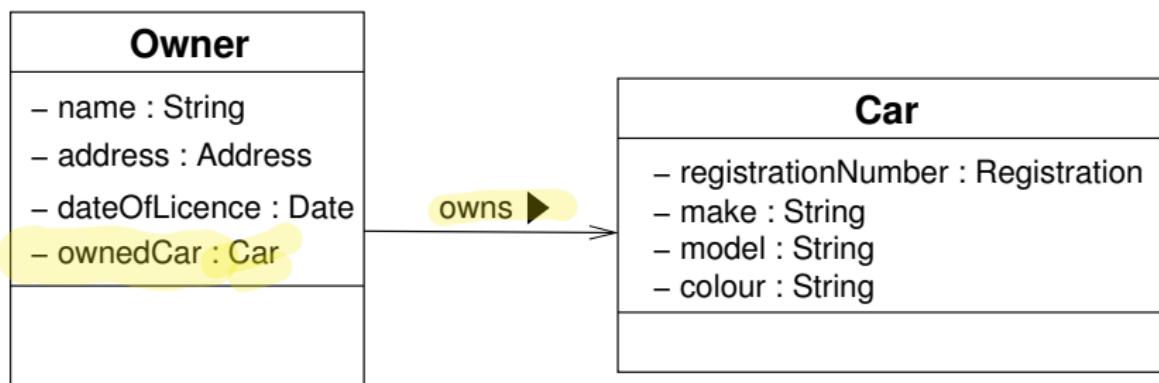


Bennett, McRobb and Farmer (2010)

Designing Associations

One-way One-to-One Association : Example

- ▶ Class Owner can send **messages** to class Car, but not vice versa.
- ▶ Each Owner object keeps a **reference** to the owned Car object.



Bennett, McRobb and Farmer (2010)

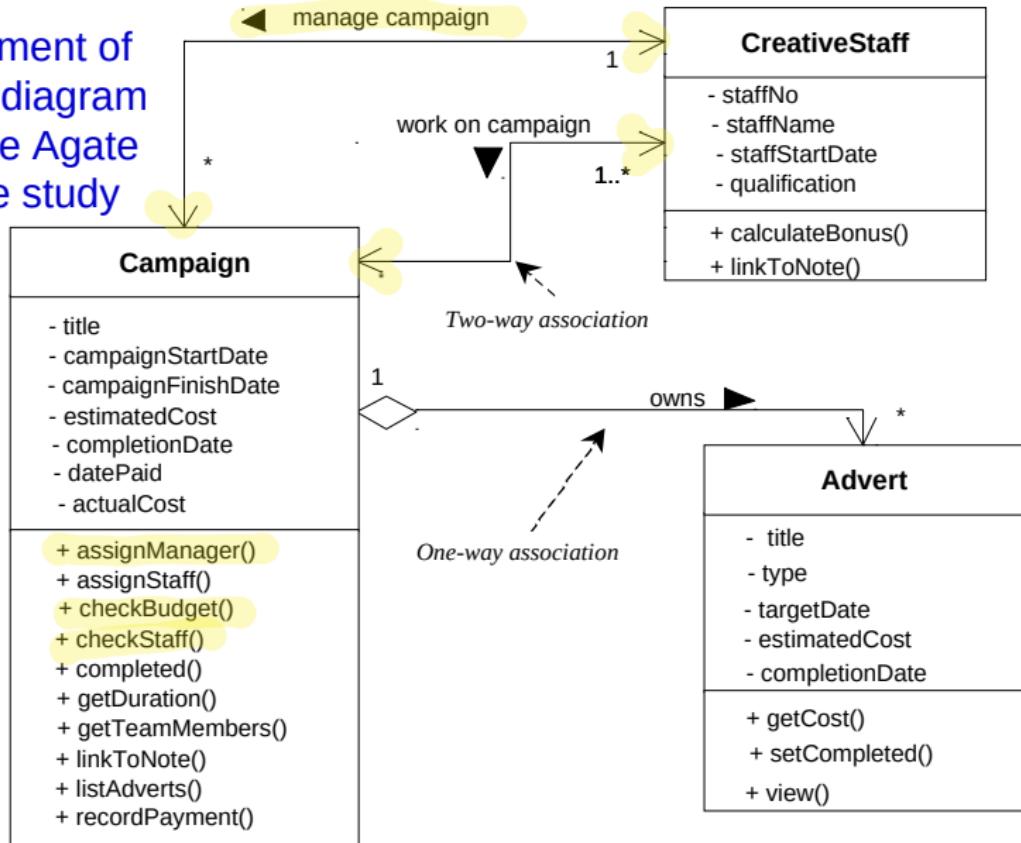
Designing Associations

Two-way Association

- ▶ An *association* that has to support message passing in both directions is a **two-way association**.
- ▶ A two-way association is indicated with arrowheads at both ends, where the arrowheads specify the permissible direction of *navigation*.
 - Loosely put, *coupling* describes the *degree of interconnectedness* between classes.
 - Traditionally, a good design seeks to minimise *coupling*.
 - *Minimizing* the number of two-way associations keeps the *coupling* between objects as *low* as possible.

One-way and Two-way Associations : Example

Fragment of class diagram for the Agate case study



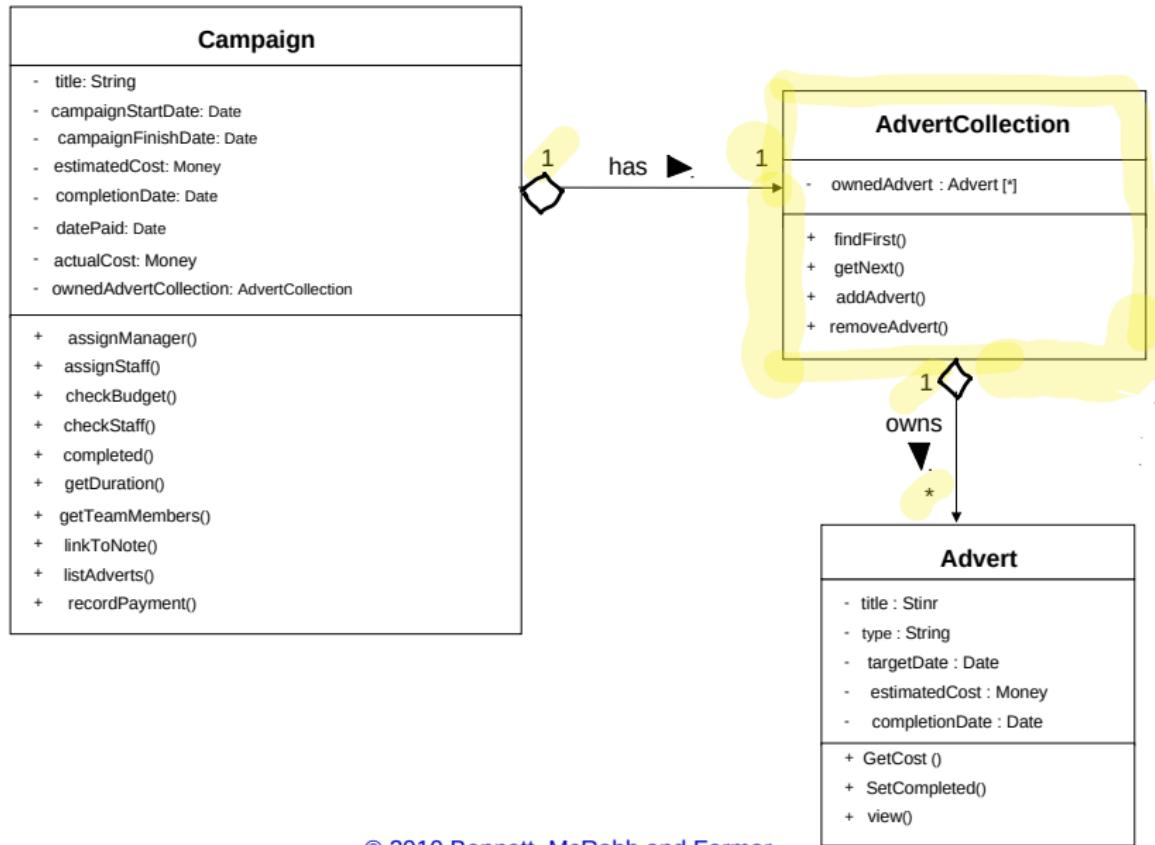
Designing Associations

Collection Classes

- ▶ *Collection* classes are used to hold the object identifiers (i.e. object references) when message passing is required *from one to many* along an association.
What does **message passing** in the above mean?
- ▶ OO languages typically provide support for working with *collection* classes.
E.g. in the Agate case study, an advertising campaign may include > 1 advertisement. Class Campaign may include a field such as:

```
private List<Advert> adverts;
```

One-to-many association using a collection class



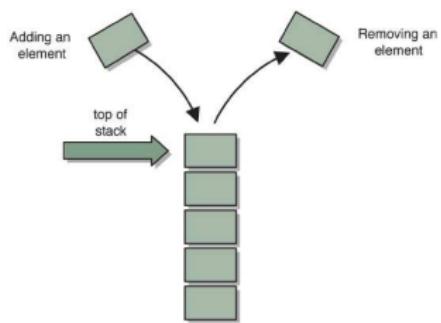
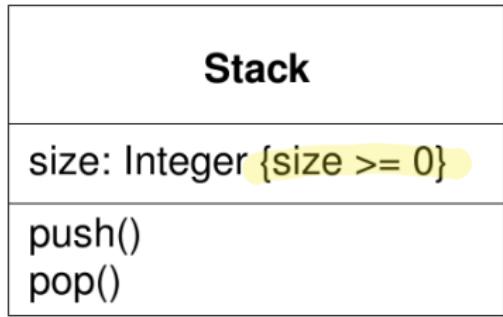
© 2010 Bennett, McRobb and Farmer

Integrity Constraints

- **Referential Integrity** ensures that an **object identifier** (object reference) in an object is actually referring to an object that **exists**, otherwise, it should refer to the **null** value.
E.g.: the Campaign object reference(s) which a CreativeStaff object keeps must be either **null** (i.e. the staff is not working on any campaign) or existing Campaign object reference(s).
- **Dependency Constraints** ensures that **attribute dependencies**, where one attribute may be calculated from other attributes, are maintained **consistently**.
An attribute whose value is calculated from other attributes is known as a **derived attribute** and is marked by / in UML (cf p.4).
- **Domain Integrity** ensures that attributes only hold **permissible values**.
E.g.: attributes from the Cost domain must be non-negative recorded in 2 decimal places.

Integrity Constraints

- ▶ Example:



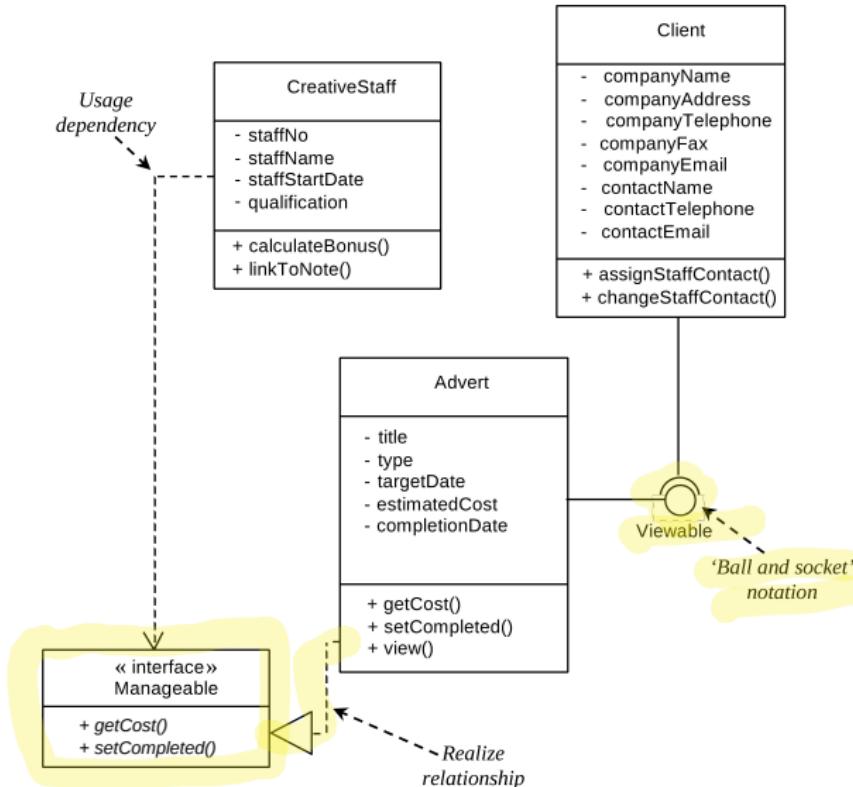
Interfaces

Interface Specification : UML Notations

- ▶ UML supports two notations to show *interfaces*:
 - ▶ The *small circle* icon showing no detail.
 - ▶ A *stereotyped class* icon with a list of the operations supported.
 - Normally only *one* of these is used on a diagram.
- ▶ The *realization* relationship, represented by the dashed line with a (non-filled) triangular arrowhead, indicates that the *client* class supports at least the operations listed in the interface.

Interfaces

Interface Specification : Examples



UML Class Diagram: Exercise

Draw an UML class diagram to describe the following class relations:

- ▶ Interface: Speaker
- ▶ Classes: Dog, Philosopher, Seminar, Conference and GUIConference
- ▶ Classes Dog and Philosopher relate to Speaker by a *realisation* relation.
- ▶ A Conference object has many Seminar objects.
- ▶ A Seminar object may only exist within the context of a Conference.
- ▶ A GUIConference object obtains seminar speech contents from a Conference object for display.

UML Class Diagram: Exercise

Draw your UML class diagram here...

Cohesion and Coupling

What are they?

- ▶ Bennett, McRobb and Farmer (2010):
 - ▶ *Cohesion is a measure of the degree to which an element contributes to a single purpose.*
 - ▶ *Coupling describes the degree of interconnectedness between design components. It is reflected by the number of links an object has and by the degree of interaction the object has with other objects.*
- ▶ Braude and Bernstein (2011):
 - ▶ *Cohesion within a module¹ is the degree to which the module's elements belong together... it is a measure of how focused a module is.*

Cohesion and Coupling

What are they? (Bennett et al., 2010)

- ▶ The concepts of coupling and cohesion are not mutually exclusive but actually *support* each other.
- ▶ Traditional detailed design tried to *maximise cohesion* elements of module of code so all contribute to the achievement of a single function.
- ▶ Traditional detailed design tried to *minimise coupling* - unnecessary linkages between modules that made them difficult to maintain or use in isolation from other modules.

Cohesion and Coupling Types

- ▶ Several ways in which coupling and cohesion can be applied within an *object-oriented approach*:
 - ▶ Interaction coupling
 - ▶ Inheritance coupling
 - ▶ Operation cohesion
 - ▶ Class cohesion
 - ▶ Specialization cohesion
 - ▶ Temporal cohesion

Coupling

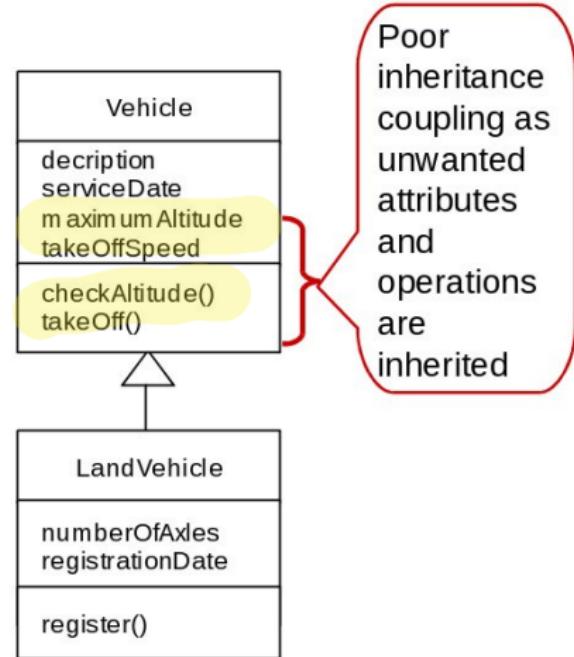
Interaction Coupling

- ▶ *Interaction Coupling* is a measure of the number of *message types* an object sends to other objects and the number of *parameters* passed with these message types.
- ▶ As a rule of thumb:
"In general, if a Message Connection involves more than three parameters, examine it to see if it can be simplified."
(Coad and Yourdon, 1991, Section 8.2.1)
- ▶ Interaction coupling in a detailed design should be kept to a *minimum* to reduce the possibility of changes rippling through the interfaces and to make reuse easier.

Coupling

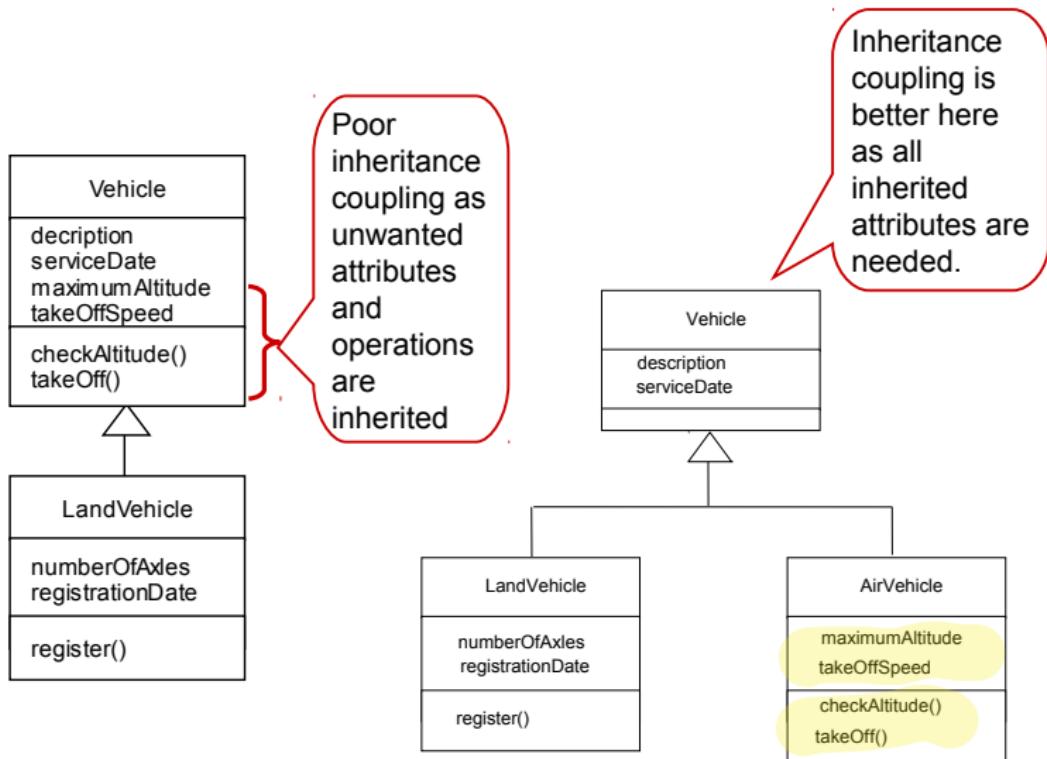
Inheritance Coupling

- ▶ *Inheritance Coupling* describes the degree to which a **subclass** actually needs the features it inherits from its **base class**.
- ▶ As a rule of thumb, class inheritance should not be used in abundance or carelessly as it may weaken the degree of *information hiding* in the classes concerned.



© 2010 Bennett, McRobb and Farmer

Inheritance Coupling – Example (Bennett et al., 2010)



Coupling

Inheritance Coupling : Another example

java.util

Class Stack<E>

```
java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractList<E>
            java.util.Vector<E>
                java.util.Stack<E>
```

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

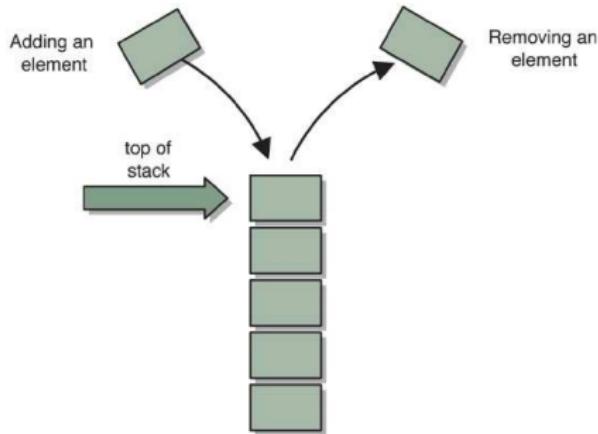
Methods inherited from class java.util.Vector

```
add, add, addAll, addAll, addElement, capacity, clear, clone, contains,
containsAll, copyInto, elementAt, elements, ensureCapacity, equals,
firstElement, forEach, get, hashCode, indexOf, indexOf,
insertElementAt, isEmpty, iterator, lastElement, lastIndexOf,
lastIndexOf, listIterator, listIterator, remove, remove, removeAll,
removeAllElements, removeElement, removeElementAt, removeIf,
removeRange, replaceAll, retainAll, set, setElementAt, setSize, size,
sort, spliterator, subList, toArray, toArray, toString, trimToSize
```

Coupling

Inheritance Coupling : Another example

- Classes `Vector` and `Stack` in package `java.util` is an example of *poor* inheritance coupling because `Stack` inherits a range of unsuitable methods from `Vector`.



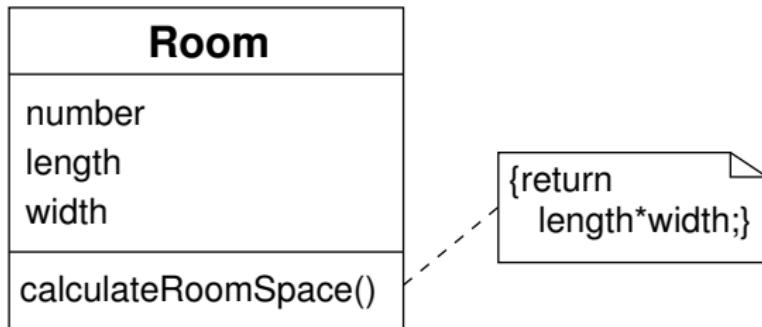
► `Stack` is a Last-In-First-Out (LIFO) collection with access to its elements at the **top** of the stack only.

► Class `java.util.Stack` inherits inappropriate methods such as `insertElementAt`, `firstElement`, `setElementAt`, `removeElementAt` and `sort` from `java.util.Vector`.

Cohesion

Operation Cohesion (Bennett et al., 2010)

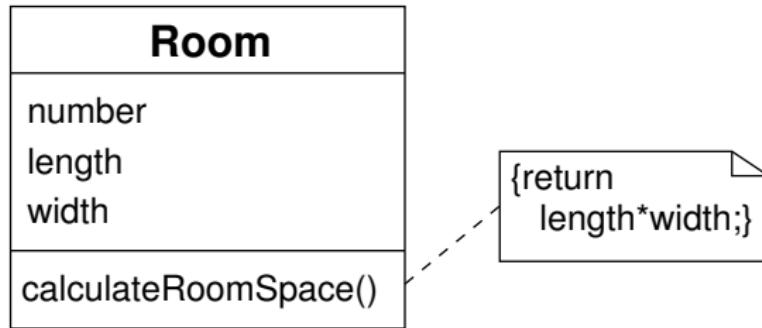
- ▶ *Operation cohesion* measures the degree to which an **operation** focuses on a *single functional requirement*.
- ▶ Example of *good* operation cohesion:



calculateRoomSpace() performs an atomic task: *compute the total area of the room.*

Cohesion

Operation Cohesion : Exercise



Based on the above class design, can you suggest an example of poor *operation cohesion* for the method **calculateRoomSpace()**?

Cohesion

Class Cohesion (Bennett et al., 2010)

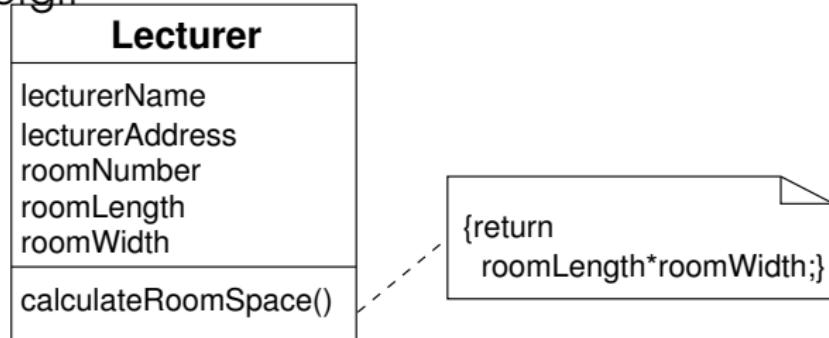
- ▶ *Class cohesion* measures the number of things a **class** represents.
 - Each class should represent **only a single entity**, e.g. lecturer, student, programme, module, etc.
- ▶ *Class cohesion* measures the degree to which a **class** is focused on a **single requirement**,
e.g. maintaining a particular type of record.
- ▶ Example of **good** class cohesion:

Lecturer
name
address
getName() getAddress() setAddress()

→ Lecturer includes attributes and operations which address the same requirement:
maintain lecturer record.

Cohesion

Good *operation cohesion* needs not entail good *class cohesion*, e.g.:



Bennett, McRobb and Farmer (2010)

Can you identify an issue with the above design?

Why?

Cohesion

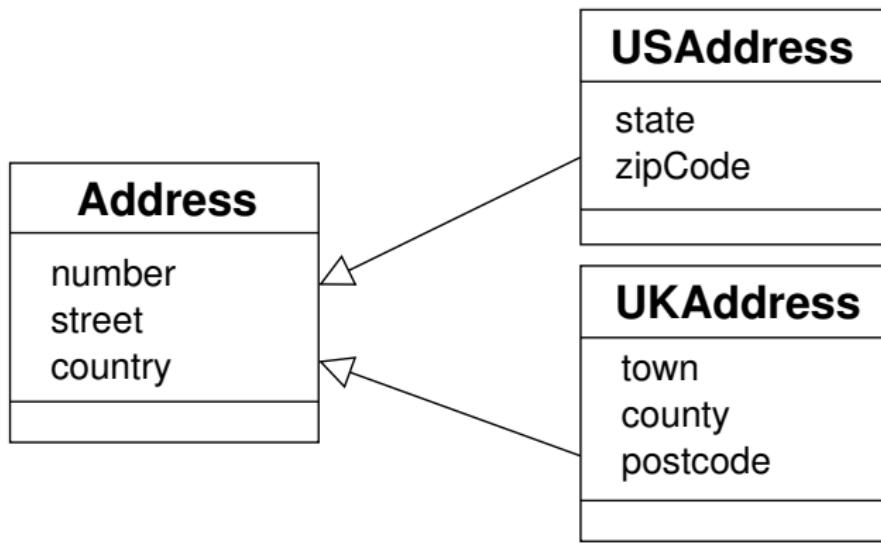
Specialization Cohesion (Bennett et al., 2010)

- ▶ *Specialization cohesion* addresses the semantic cohesion of inheritance hierarchies.
- ▶ It measures the *semantic-relatedness* between the inherited attributes and operations to the class itself.
 - I.e. are the classes related to each other through a *is_a* (or "*is_a_kind_of*") relation? Or is the generalization/specialization relation defined out of convenience?
- ▶ Class A should **extend**, and hence **inherit** attributes and operations from, class B **only** when A is a *specialized* version of B, not simply because it needs to use the attributes and operations in B.

Cohesion

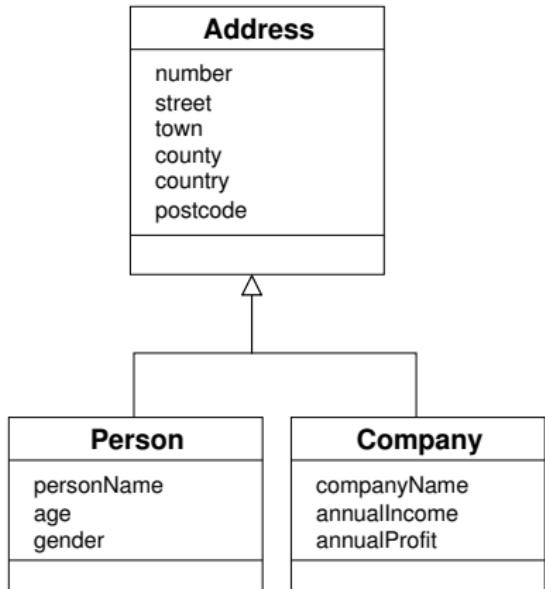
Specialization Cohesion : Example (Bennett et al., 2010)

Example of a good *specialization cohesion*:

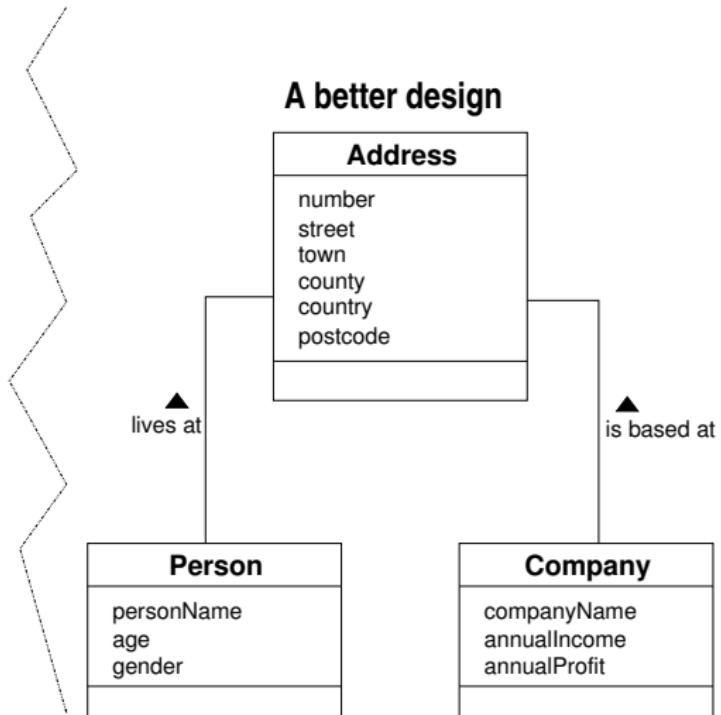


Specialization Cohesion

Poor design:
low specialization cohesion



A better design



Bennett, McRobb and Farmer (2010)

► *Temporal cohesion:*

- module elements linked only by the time at which they need to be done
- e.g.: an **initialisation** module

“Temporal cohesion is present when a subprogram performs a set of functions that are related in time, such as ‘initialisation’, ‘house-keeping’, and ‘wrap-up’. . . . The only connection between these operations is that they are performed within the same limited time-span.” p.91
Information Systems Engineering: An Introduction, By Arne Soelvberg, David C. Kung

Acknowledgement: Thanks to L. J. Hazlewood for his *ISE* (2010) slides on various types of cohesion and coupling.

Cohesion and Coupling

Points to note

- ▶ *"Low cohesion classes often represent a very 'large grain' of abstraction or have taken on responsibilities that should have been delegated to other objects. . . As a rule of thumb, a class with high cohesion has a relatively small number of methods, with highly related functionality, and does not do too much work. It collaborates with other objects to share the effort if the task is large."*

Larman (2005, pp.314–317)

- ▶ *". . . no coupling between classes . . . offends against a central metaphor of object technology. . . Low coupling taken to excess yields a poor design. . . It is not high coupling per se that is the problem; it is high coupling to elements that are unstable in some dimension, such as their interface, implementation, or mere presence."*

Larman (2005, pp.299–302)

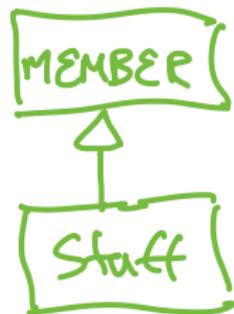
Liskov Substitution Principle

What is it? Liskov (1988, p.7)

"What is wanted here is something like the following substitution property: If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T , the behavior of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T ." Liskov (1988, p.7)

```
public void setDriver(Member carUser) {  
    if (carUser.hasDrivingLicence()) {  
        driver = carUser;  
    }  
}
```

```
public void setDriver(Staff carUser) {  
    if (carUser.hasDrivingLicence()) {  
        driver = carUser;  
    }  
}
```



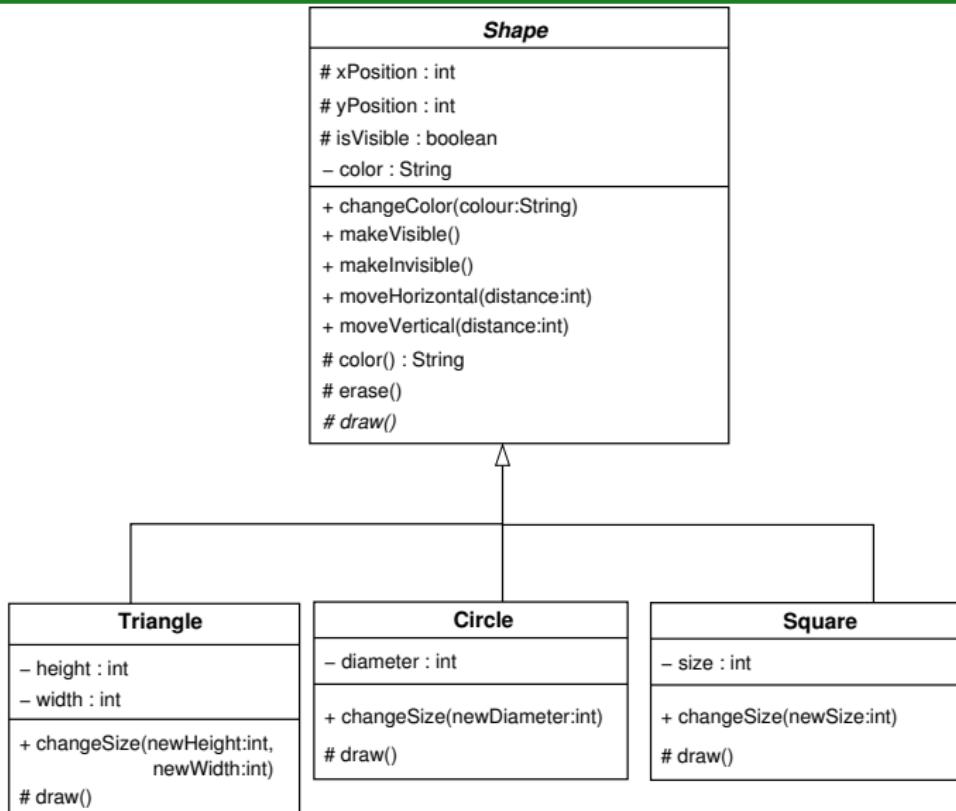
Liskov Substitution Principle

What is it?

- ▶ “*Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.*”
Martin (1996)
- ▶ Essentially the principle states that, in object interactions, it should be possible to treat a **derived object** (an instance of a **subclass**) as if it were a **base object** (an instance of a **superclass**) without integrity problems.

Liskov Substitution Principle

A Liskov-compliant Class Hierarchy



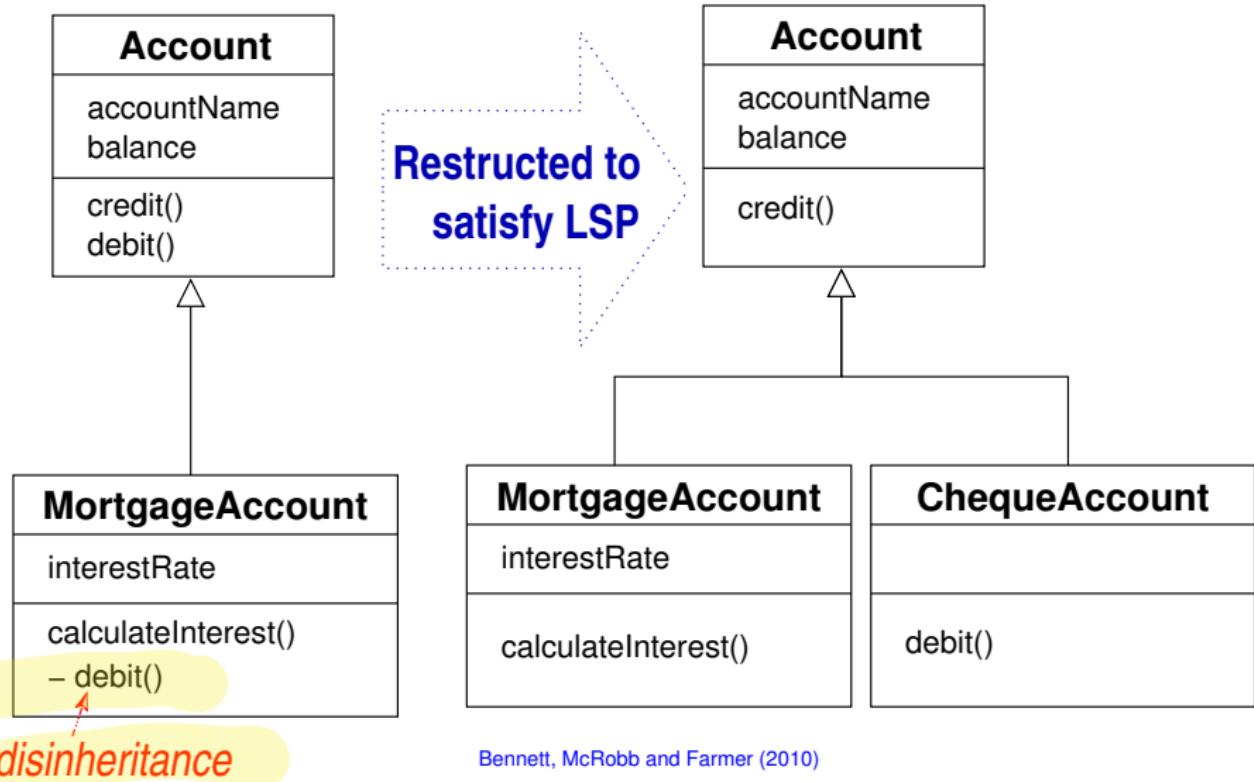
Liskov Substitution Principle

Why LSP?

- ▶ If LSP is not applied, it may be possible to violate the *integrity* of the derived object, e.g.:
- ▶ Subclasses typically inherit attributes and operations from their superclass.
- ▶ If an attribute or operation is unsuitable (not applicable) for the semantic of a subclass, it is possible for the subclass to “*disinherit*” it by declaring it **private**.
 - ➡ However, this will violate the Liskov Substitution Principle.

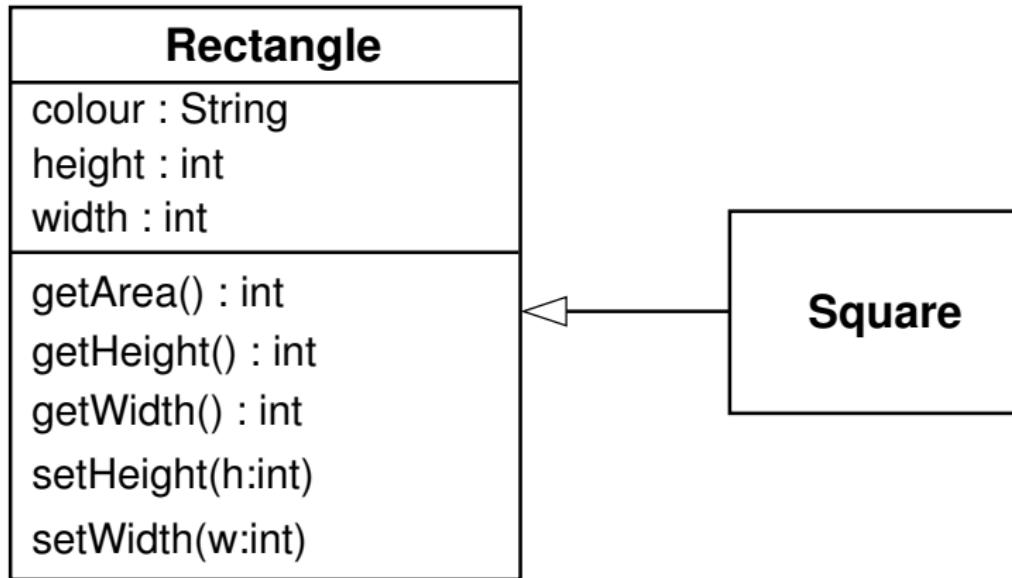
Liskov Substitution Principle

Applying Liskov Substitution Principle : Example



Liskov Substitution Principle : Exercise

Is the following class design Liskov-compliant?



Liskov Substitution Principle

Points to note

- ▶ LSP helps to ensure that an OO class design is *maintainable* and *reusable*.
- ▶ A good OO detailed design should conform with the LSP.

Detailed Class Design

To sum up:

- ▶ Check that *responsibilities* have been assigned to the right class.
- ▶ Define or use *interfaces* to group together well-defined *standard* behaviours.
- ▶ Apply the concepts of *coupling* and *cohesion*.
- ▶ Apply the *Liskov Substitution Principle*.

References

- ▶ Bennett, S., McRobb, S. and Farmer, R., *Object-Oriented Systems Analysis and Design Using UML*, 4th ed, Maidenhead: McGraw-Hill, 2010.
- ▶ Braude, E.J. and Bernstein, M.E., *Software Engineering: Modern Approaches*, 2nd ed, Hoboken, NJ: Prentice Hall, 2005.
- ▶ Coad, P and Yourdon, E., *Object-Oriented Design*, Englewood Cliffs, NJ: Yourdon Press, Prentice-Hall, 1991.
- ▶ Dennis, A., Wixom, B. H. and Tegarden, D. , *Systems Analysis and Design: An Object-Oriented Approach with UML*, 5th ed, Danvers, MA: Wiley, 2015.
- ▶ Larman, C., *Applying UML and patterns*, 3rd ed, Upper Saddle River, NJ: Wiley, 2011.

References

- ▶ Liskov, B., "Data Abstraction and Hierarchy", 1988, (Online). Available at: <http://citeseer.ist.psu.edu/viewdoc/download?doi=10.1.1.12.819&rep=rep1&type=pdf> (08/11/2017).
- ▶ Martin, R.C., "The Liskov Substitution Principle", 1996 (Online). Available at <https://web.archive.org/web/20151128004108/http://www.objectmentor.com/resources/articles/lsp.pdf> (08/11/2017).
- ▶ OMG, *OMG Unified Modeling Language™ (OMG UML), Superstructure*, Version 2.3, 2010, (Online). Available at: <http://www.omg.org/spec/UML/2.3/Superstructure/PDF/> (08/11/2017).

References

- ▶ OMG, *OMG Unified Modeling LanguageTM (OMG UML)*, Version 2.5, 2013, (Online). Available at:
<http://www.omg.org/spec/UML/2.5/Beta2/PDF>
(08/11/2017).
- ▶ Soelvberg, A. and Kung, D. C., *Information Systems Engineering: An Introduction*, Springer-Verlag, 1993.
- ▶ Yourdon, E. and Constantine, L.L., *Structured Design: Fundamentals of a Discipline of Computer Program and System Design*, Upper Saddle River, NJ: Prentice-Hall, 1979.

For detail, see:

*Chapter 14 of Bennett et al. (2010) and
Chapter 16 of Braude & Bernstein (2011).*

Learning Outcomes. You should now be able to

- ▶ specify a detailed object-oriented design using UML class diagram
- ▶ describe some criteria for good object-oriented detailed design
- ▶ evaluate a detailed object-oriented design
- ▶ specify classes, attributes, operations, relations, comments and constraints in UML
- ▶ state what is meant by LSP and use LSP to evaluate a detailed design

Unit 10 Object-Oriented Design Patterns

Unit Outcomes. Here you will learn

- ▶ some well-known design patterns:
 - ▶ Singleton
 - ▶ Adapter
 - ▶ State
 - ▶ Strategy
 - ▶ Observer
 - ▶ Composite

Further Reading: Larman (2005) Ch26, Braude & Bernstein (2011) Ch17

based on Bennett, McRobb & Farmer (2010) and Braude & Bernstein (2011)

Patterns

What is a design pattern?

- ▶ A design pattern is a “*description of communicating objects and classes that are customized to solve a general design problem in a particular context*”.

Gamma et al. (1995, p.3)

- ▶ Patterns are *problem-centred*, not *solution-centred*.
- ▶ Patterns capture and communicate “*best practice*” and expertise.

Patterns

Patterns and Non-functional Requirements

- ▶ Patterns are typically applied to address **non-functional requirements**, which include:
 - ▶ changeability
 - ▶ interoperability
 - ▶ efficiency
 - ▶ reliability
 - ▶ testability
 - ▶ reusability

Buschmann et al. (1996)

Pattern Template

Bennett et al. (2010) & Gamma et al (1995)

- ▶ *meaningful Name* reflecting the knowledge embodied by the pattern.
- ▶ a description of the **Problem** that the pattern addresses.
- ▶ **Context**- the (*circumstances* in which the pattern can be applied.

Pattern Template

► **Forces:**

- the *constraints* addressed by the solution.
- Any trade-offs or language-specific issues.

► **Solution:**

- the components involved in the pattern and their relationships.
 - May be depicted as a UML *class diagram* or *object diagram*.

GoF Design Patterns

- ▶ Gamma et al. (1995) documented a catalogue of 23 design patterns.
 - ➡ *GoF (Gang of Four)* refers to the authors of the book.
- ▶ *GoF Patterns* are classified by two criteria:
 - ▶ **purpose:** *creational, structural & behavioural*
 - ▶ **scope:** *class & object*

GoF Design Patterns

Creational Patterns

- ▶ Concerned with the *construction* of object instances.
- ▶ Separate the operation of an application from how its objects are created.
- ▶ Examples:
 - ▶ Class: *Factory*
 - ▶ Object: *Abstract Factory, Builder, Prototype, Singleton*

GoF Design Patterns

Structural Patterns

- ▶ Concerned with the way in which classes and objects are organized.
- ▶ Offer effective ways of using object-oriented constructs such as inheritance, aggregation and composition to satisfy particular requirements.
- ▶ Examples:
 - ▶ Class: **Adapter**
 - ▶ Object: **Adapter**, *Bridge*, **Composite**, *Decorator*, *Façade*, *Flyweight*, *Proxy*

GoF Design Patterns

Behavioural Patterns

- ▶ Address the problems that arise when assigning responsibilities to classes and when designing algorithms.
- ▶ Suggest particular static relationships between objects and classes and also describe how the objects communicate.
- ▶ Examples:
 - ▶ Class: *Interpreter, Template Method*
 - ▶ Object: *Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy, Visitor*

Behavioural Pattern: Observer

Context/Issue

- ▶ An investment company helps clients find profitable stocks in the market for investment.
 - ▶ The company keeps an account for *each* client.
 - ▶ A client may be investing in *multiple* stocks.
 - ▶ The same stock is typically being purchased by *multiple* clients.
- ▶ The price of a stock changes *continually*.
- ▶ The investment company needs to maintain *up-to-date* stock values for all clients.
- ▶ *How to ensure that when stock value changes, the clients' accounts are updated immediately?*

Behavioural Pattern: Observer

Potential Solutions : 1

1. Equip class Stock with an **update** method which, when invoked, updates the respective value in all **Account** objects which are referenced by it.

```
1 public class Stock {  
2     private double value;  
3     private Set<Account> owners;  
4     // other fields, constructor and methods omitted  
5  
6     private void updateOwners() {  
7         for(Account owner : owners) {  
8             owners.changeStockValue(this, value);  
9         }  
10    }  
11 }
```

What are the drawbacks of this approach?

Behavioural Pattern: Observer

Potential Solutions : 2

2. *"Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically."*

Gamma et al. (1995, p.293)

→ The **Observer** pattern...

Behavioural Pattern: Observer

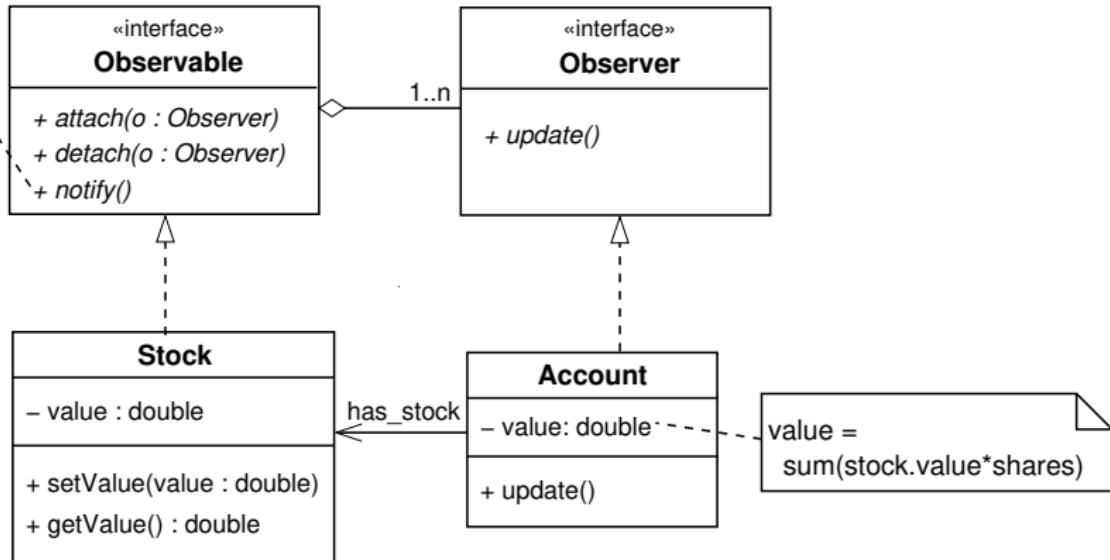
Solution : description

- ▶ Enable the object acting as the *data source* to communicate with *all* objects which use the data (i.e. the **Observer** objects).
- ▶ Notify the **Observer** objects of any change in the data source.
- ▶ Upon receiving a change notification, each **Observer** object updates its **state**.
 - ⇒ The *observer* pattern promotes good object-oriented design because it enables the observers to keep their own data up-to-date.

Behavioural Pattern: Observer

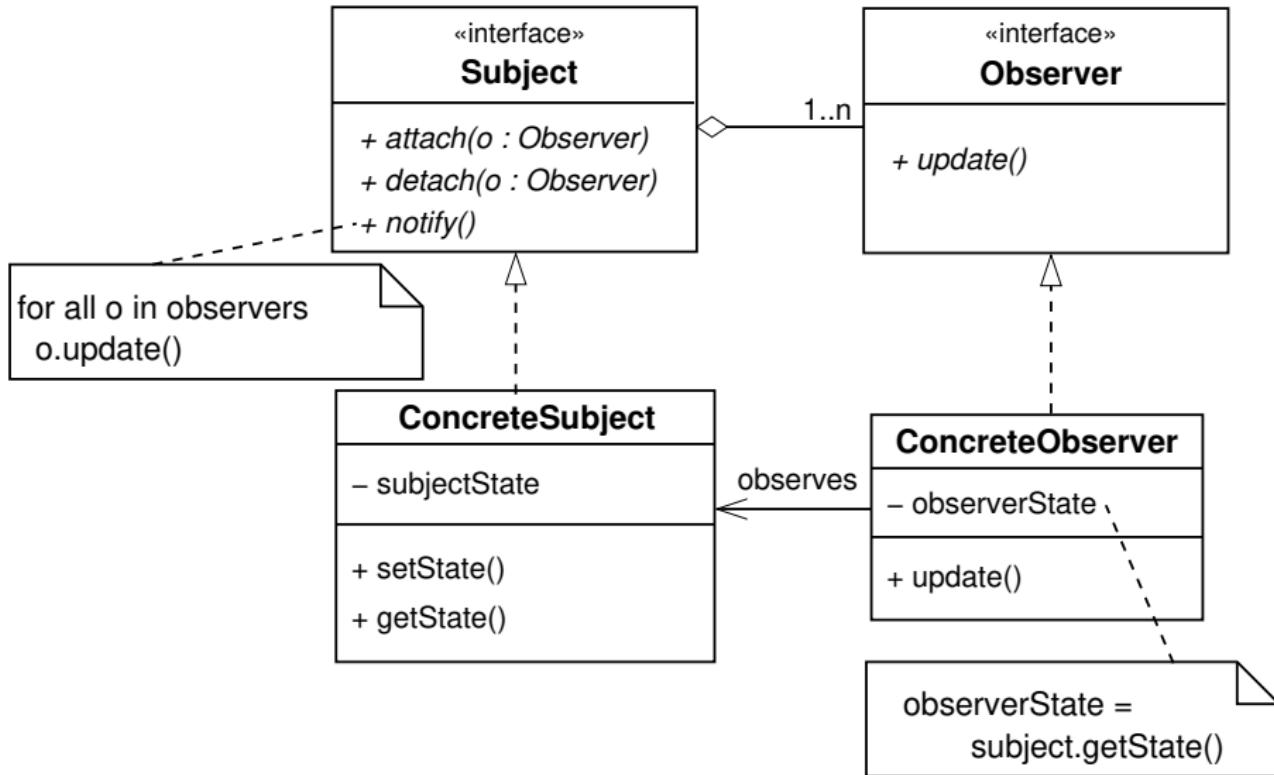
Solution : class design

```
for all o in observers  
o.update()
```



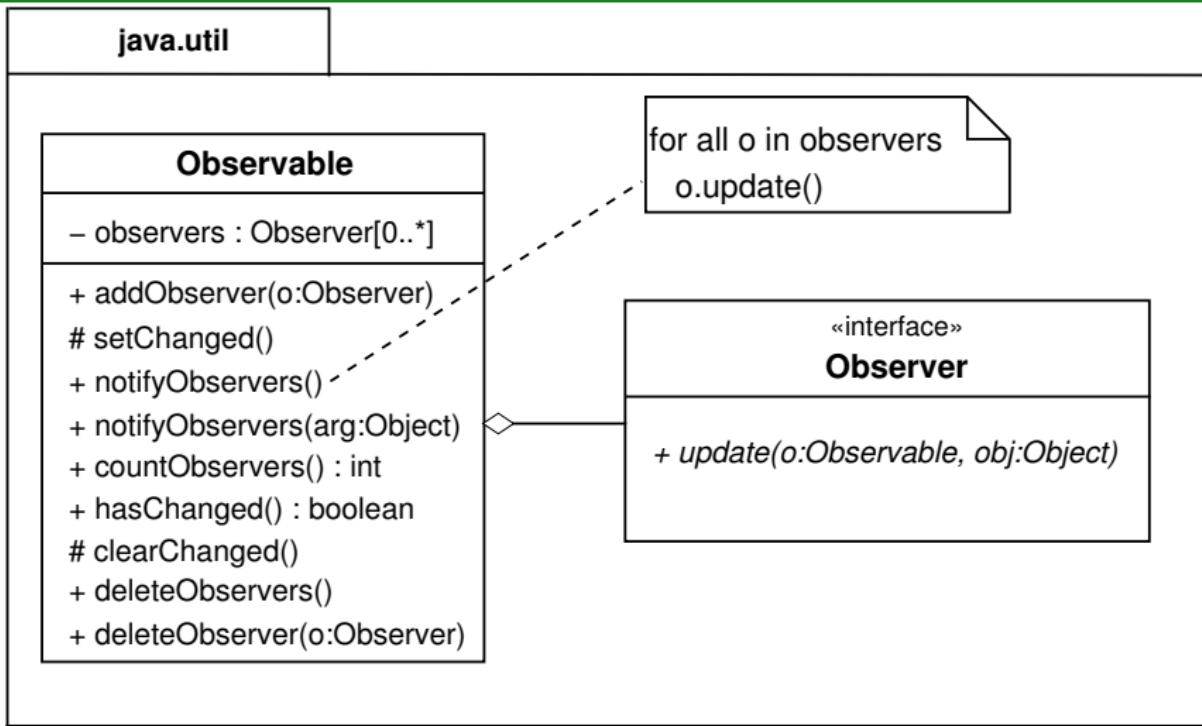
Behavioural Pattern: Observer

General form



Behavioural Pattern: Observer

An Implementation (deprecated!) in Java :
Observer and *Observable* in `java.util`



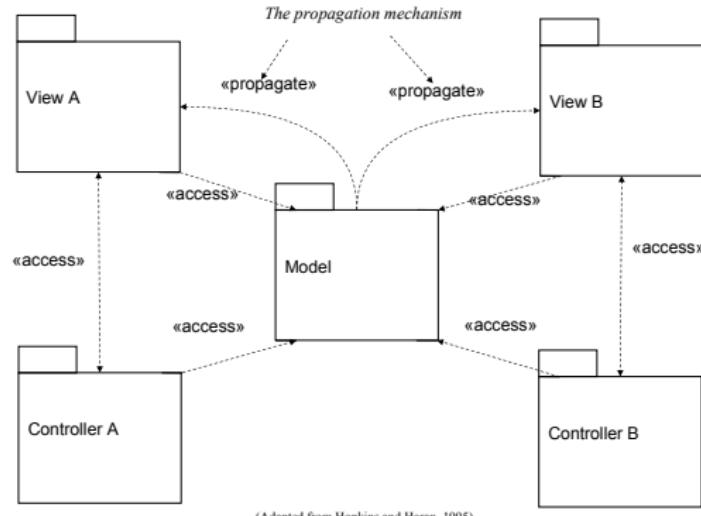
Behavioural Pattern: Observer

An Implementation (deprecated!) in Java

- ▶ Interface `java.util.Observer` and class `java.util.Observable` modelled the two key components in the *observer* pattern.
- ▶ Classes **implementing** the interface `Observer` needed to implement the inherited `update` method.
 - ⇒ Called whenever an object is observed (i.e. `Observable`) changes its `state`.
- ▶ Classes modelling data sources for observation **extended** class `Observable`, and included methods such as `addObserver`, `deleteObserver` and `notifyObservers`.
 - ⇒ When an `Observable` object changed its `state`, method `notifyObservers` needed to be called in order to notify its `Observers` to update their states.

Behavioural Pattern: Observer Discussion

- The *observer* pattern is used in the *Model-View-Controller* (MVC) architecture in which the **views** play the role of the **observer** and the **model** is the **observable**, with the **controllers** acting as **clients**, i.e. updating the observables.



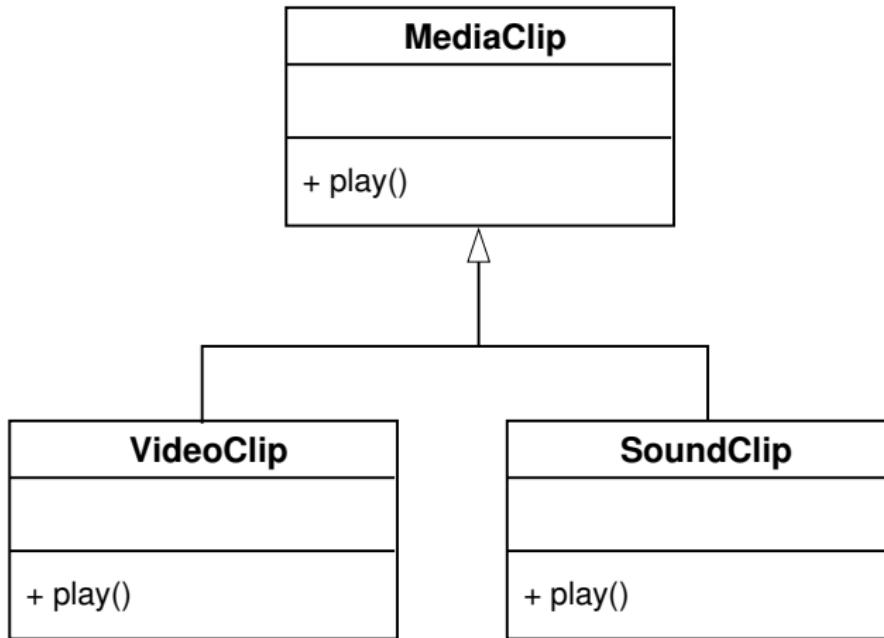
Structural Pattern: Composite

Context/Issue

- ▶ Consider the **Agate** case study: we need to store information about *media clips* for each *advertisement*...
- ▶ Each advertisement may be made up of a *single* or a *composite sequence* of media clip(s).
- ▶ Each media clip in turn may take the form of a video clip or a sound clip, or a combination of several media clips.
- ▶ How to present the *same interface* for a media clip whether it is composite or not?
- ▶ How can we incorporate *composite structures*?

Structural Pattern: Composite Context/Issue

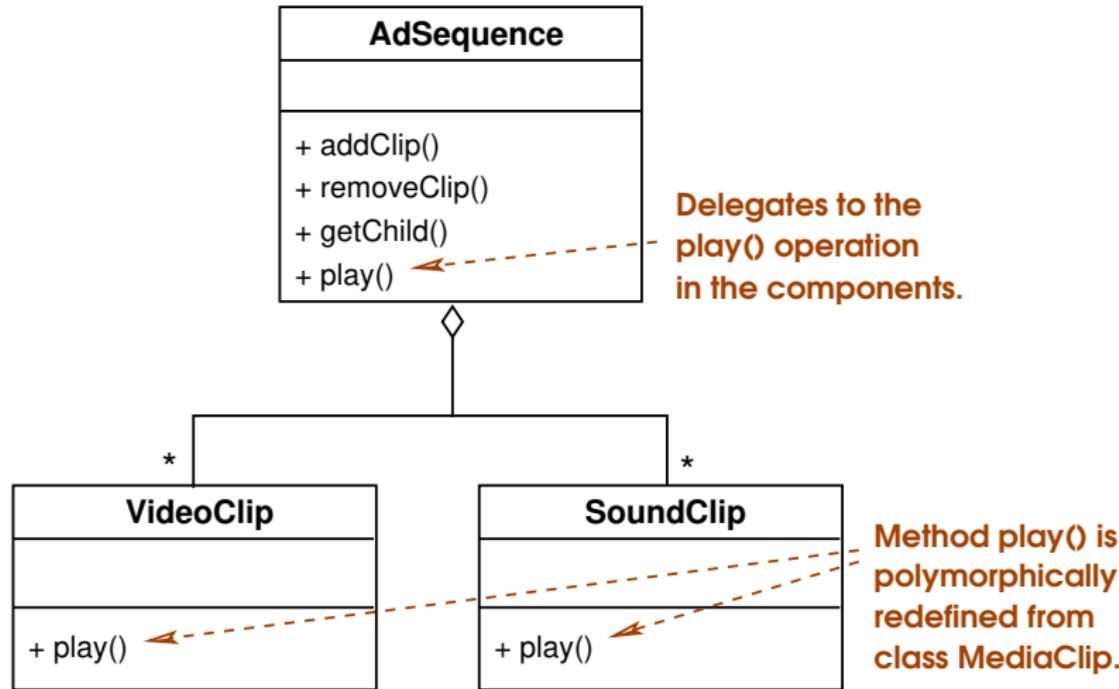
How to present the *same interface* for a media clip whether it is composite or not?



Structural Pattern: Composite

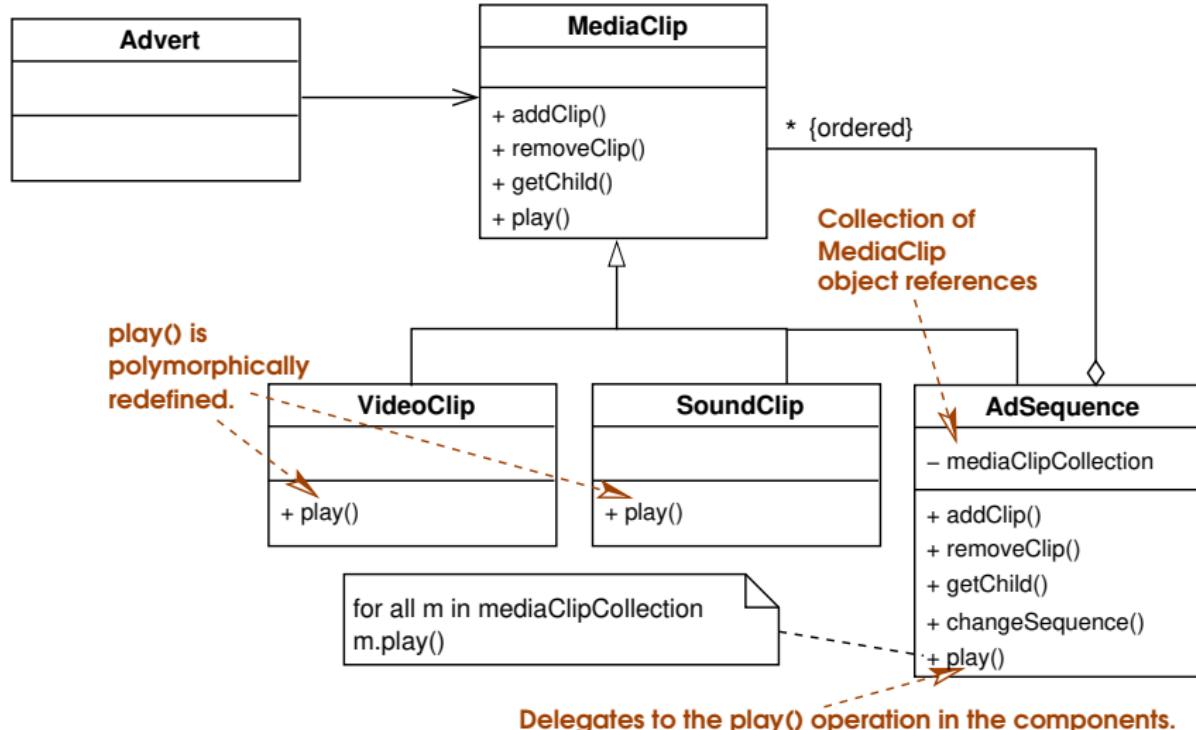
Context/Issue

How can we incorporate *composite structures*?



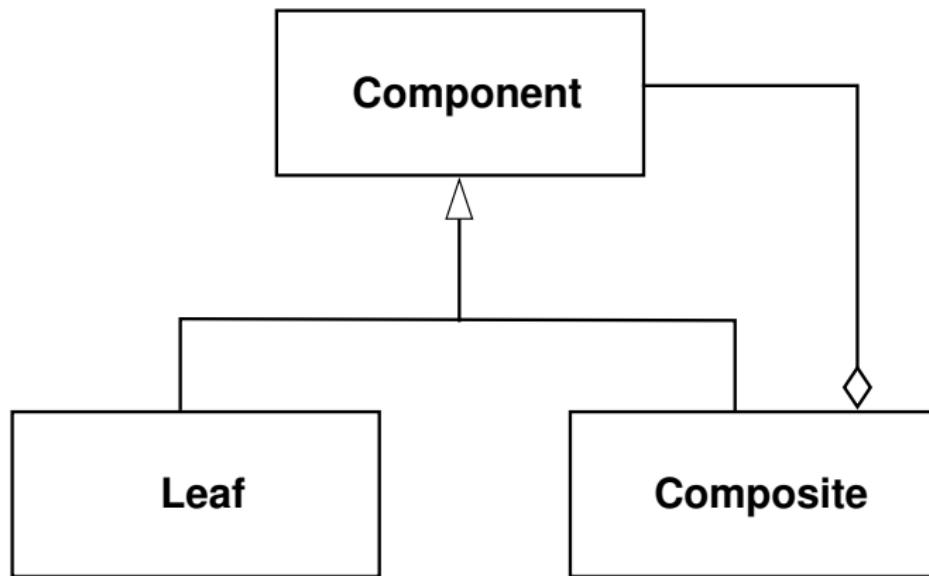
Structural Pattern: Composite

Combine inheritance and aggregation hierarchies:



Structural Pattern: Composite

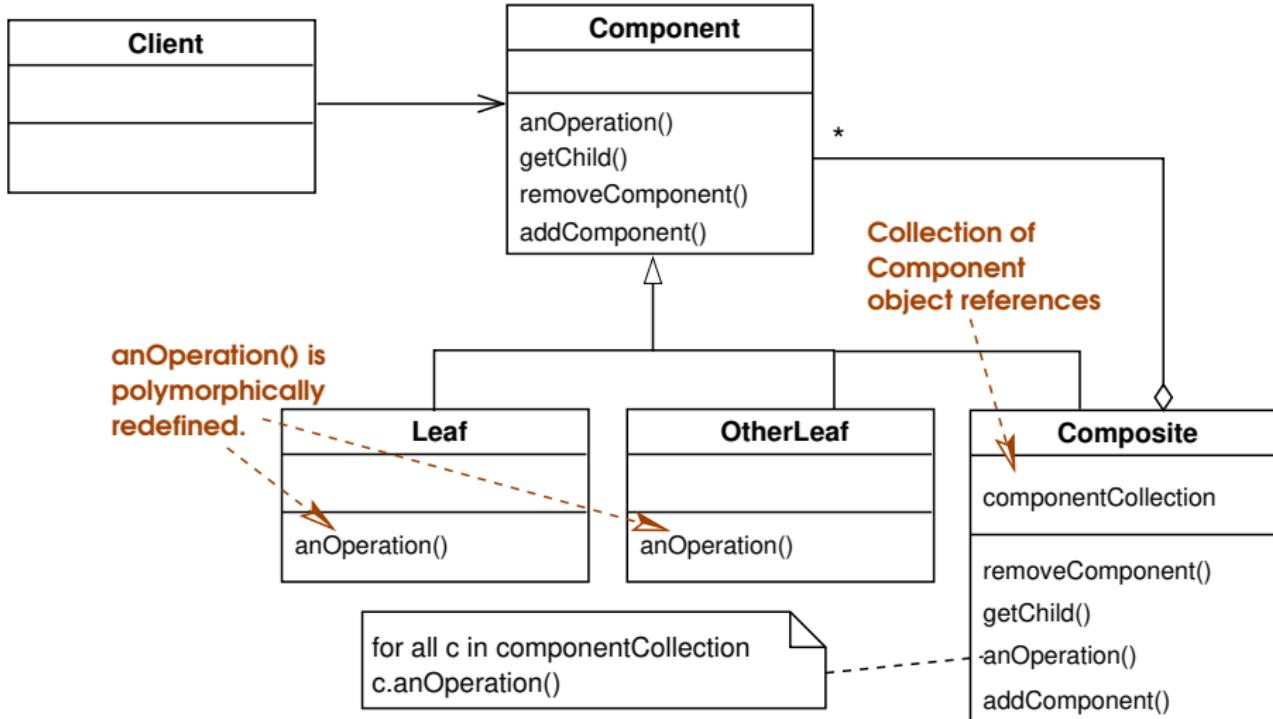
Basic Structure



2010 Bennett, McRobb & Farmer

Structural Pattern: Composite

General form



Structural Pattern: Composite

Discussion : 1

Consider the *general form* of the *Composite* pattern:

- ▶ The *Composite* pattern is designed to model *recursive tree structure* to capture a complex *part-whole relation* in a class design.
- ▶ The *Component* may be defined as an *interface*, an *abstract class* or a *concrete class*, depending whether it has attributes and operations that need to be inherited.

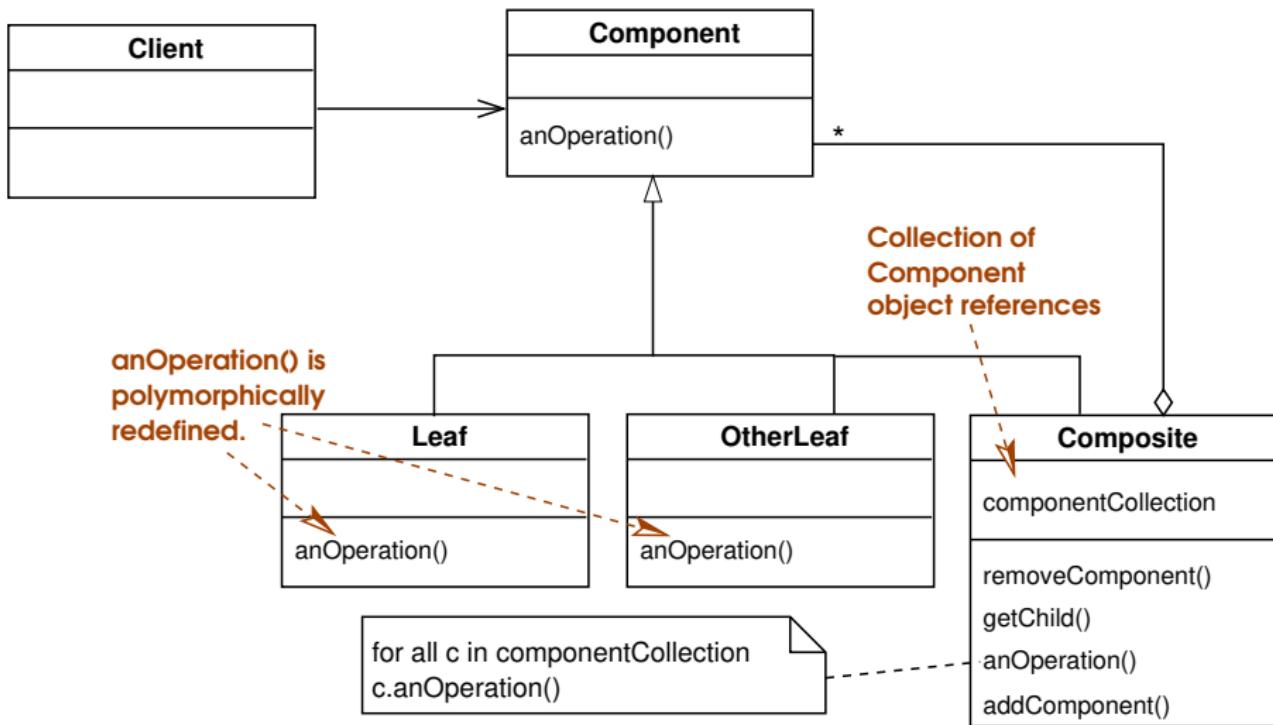
Structural Pattern: Composite

Discussion : 2

- ▶ When applying the *Composite* pattern to Java software development, the *Composite* pattern poses an issue:
Methods addComponent, removeComponent and getChild are relevant to the Composite, but irrelevant to the Leaf classes. Hence, it displays a poor level of inheritance coupling and it also violates the Liskov Substitution Principle.
- ▶ A working solution would be to define the attributes and operations that are *common* to all the class hierarchy in the *Component* class and define *addComponent*, *removeComponent* and *getChild*, which are relevant to the *Composite* only, in the *Composite* class.

Structural Pattern: Composite

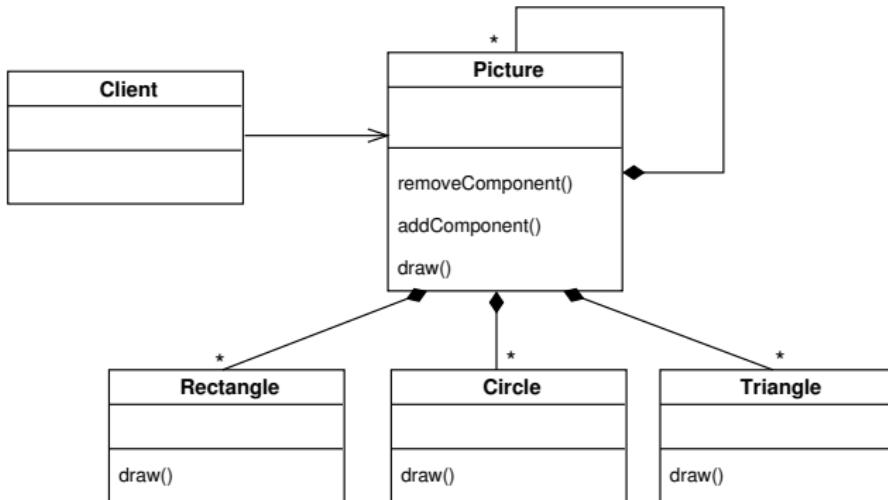
Corrected general form



Structural Pattern: Composite

Composite Pattern Vs Composite Structure

The part-whole relationship could alternatively be modelled by a simple *composition* relationship:



*What are the advantages of using the **composite pattern** instead of straight-forward composition?*

Composite Pattern Vs Composite Structure : Advantages of using composite pattern

- ▶ *Providing more flexibility:*

A client class can collaborate with a part (e.g. Component) or a whole (e.g. Composite) *in the same way* because they have the same interface.

- ▶ *Enabling reuse, reducing code duplication:*

- ▶ Common properties can be defined in the super-class and inherited by both Leaf and Composite subclasses.
- ▶ E.g.: in a graphics application, each graphic component/composite may keep an attribute `isVisible` to indicate if it should be hidden from the view or not.

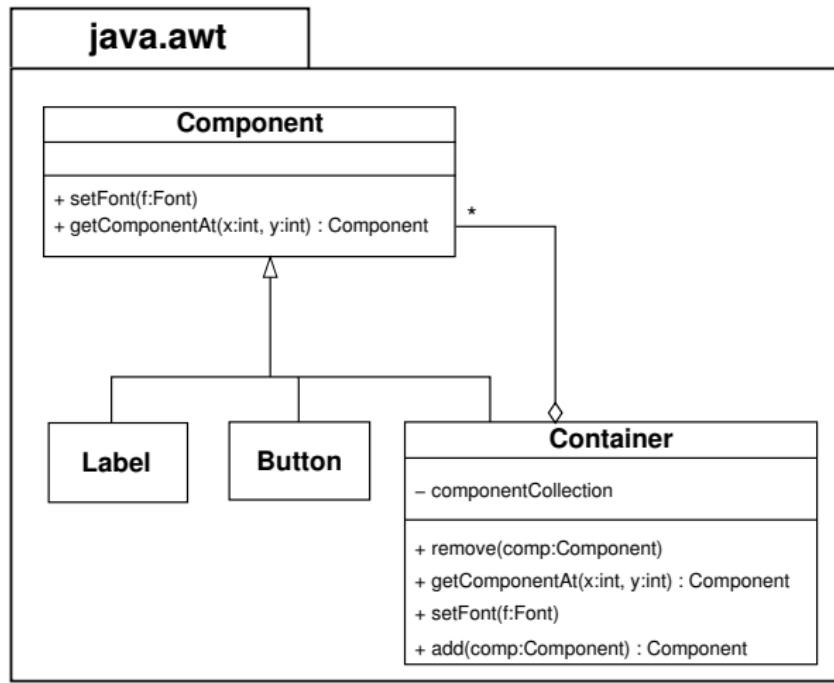
Composite Pattern Vs Composite Structure : Advantages of using composite pattern

- ▶ *Simplify the implementation by avoiding the need of type casting:*
- ▶ In the composite *structure* shown earlier, the *Composite* class keeps a reference to all of its components, in a Collection.
- ▶ However, such a collection object will need to be defined of a general type, eg *Object* in Java, so as to accommodate various types of component objects.
- ▶ *Type casting* will therefore be unavoidable when operations specific to each type of component need to be carried out.

Structural Pattern: Composite

Example from Java API

- The *composite* design pattern is used in modelling the part-whole relationship between GUI components.



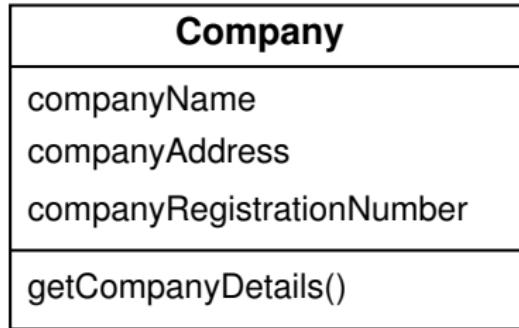
Structural Pattern: Composite

Example from Java API

```
1  /** Set the font for a component and all of
2   *      its child components to the specified font.
3   * @param component    A GUI component
4   * @param font    A font to be used within a GUI component
5   */
6  public static void setFont4All(Component component, Font font)
7  {
8      component.setFont(font);
9
10     /* A component may contain other components
11      (e.g. a JPanel object containing JButton objects) */
12     if (component instanceof Container) {
13         /* The given component is a Container object...
14             performs type casting. */
15         Container parent = (Container) component;
16
17         for (Component child : parent.getComponents()) {
18             setFont4All(child, font); // recursion
19         }
20     }
21 }
```

Creational Pattern: Singleton Context/Issue

- ▶ Consider the **Agate** case study: we need to store information about Agate Company to be accessed by different system objects.
- ▶ How to ensure that *only one* instance of the **Company** class is created?

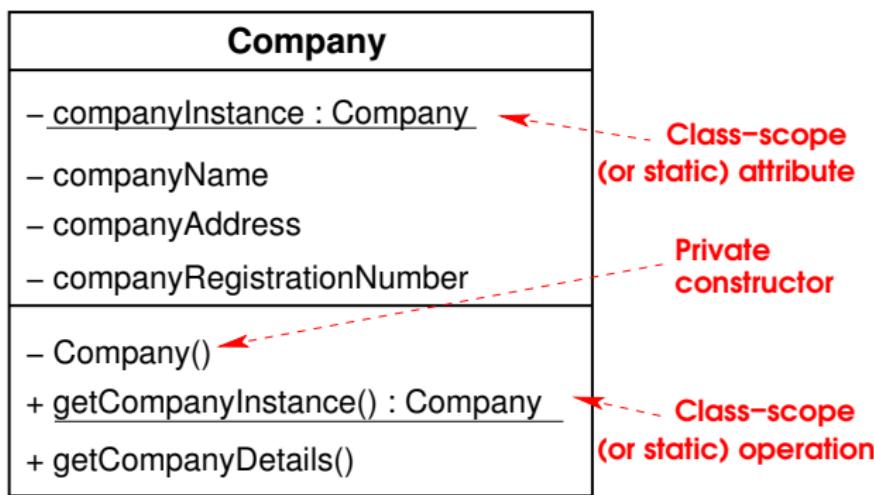


2010 Bennett, McRobb & Farmer

Creational Pattern: Singleton

Restrict access to the **constructor**.

- ▶ Use class-scope attribute (i.e. **class/static** variable) to ensure a *single instance*.
- ▶ Use class-scope operations (i.e. **class/static** methods) to allow *global access*.



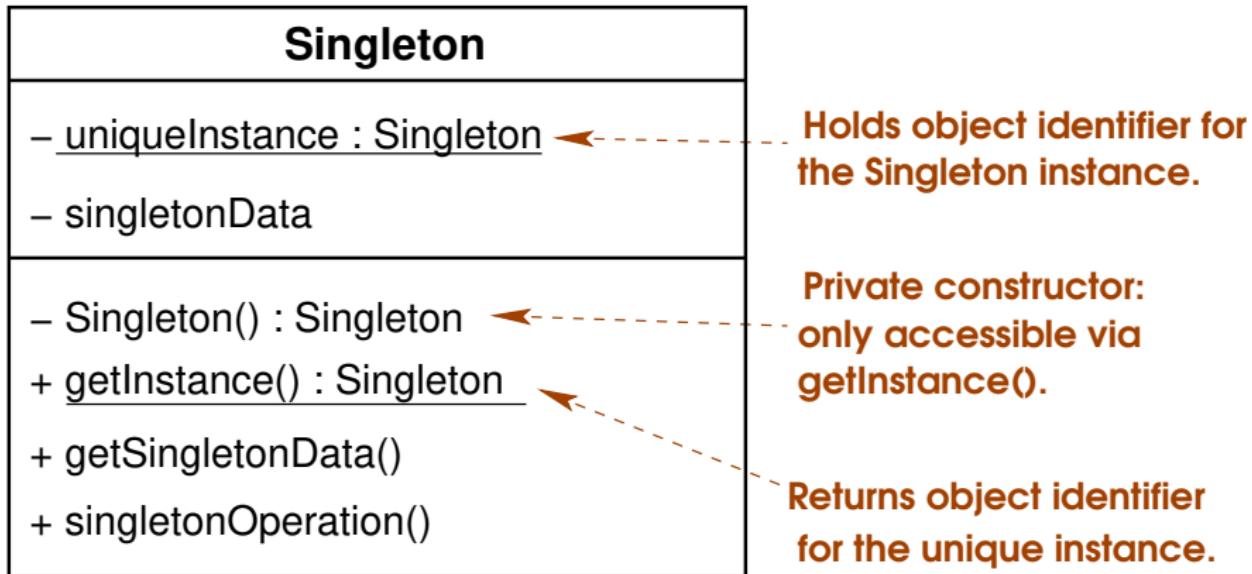
Creational Pattern: Singleton

Restricted Access to Constructor

```
1 public class Company {  
2  
3     private static Company companyInstance;  
4     // other field declaration omitted  
5  
6     private Company() {  
7         // initialise instance variables  
8     }  
9  
10    public static Company getCompanyInstance() {  
11        if(companyInstance == null) {  
12            companyInstance = new Company();  
13        }  
14        return companyInstance;  
15    }  
16  
17    // other class detail omitted  
18 }
```

Creational Pattern: Singleton

General form



2010 Bennett, McRobb & Farmer

Creational Pattern: Singleton

Examples from Java API

- ▶ The **singleton** design pattern is fairly commonly used.
Examples of such in the Java API include:
 - ▶ `java.awt.Desktop.getDesktop`
 - ▶ `java.lang.Runtime.getRuntime`

Behavioural Patterns: State Context/Issue

- ▶ Consider the class Campaign.
- ▶ It has four states:
Commissioned, Active, Completed and Paid.
- ▶ A Campaign object has different behaviour, depending on the *state* it is at.
- ▶ For example, regardless of the state of a Campaign object (i.e. *Commissioned, Active*, etc), there is the need to calculate the costs incurred by this campaign so far. Each state, however, entails a different costing model.
- ▶ *How can an object's behaviour change at run-time, based on the state it is at?*

Behavioural Patterns: State

Potential Solutions : using nested if statements

*How can an object's behaviour change at run-time according to the **state** it is at?*

1. The state-dependent operations may be implemented using `case` statements or nested `if` statements to specify alternative behaviour.

Behavioural Patterns: State Potential Solutions : using nested if statements

«entity»	
Campaign	
- title	
- campaignStartDate	
- campaignFinishDate	
- estimatedCost	
- completionDate	
- datePaid	
- actualCost	
- campaignOverheads	
- advertCollection	
- teamMembers	
+ Campaign()	
+ assignManager()	
+ assignStaff()	
+ checkCampaignBudget()	
+ calcCosts()	-----
+ checkStaff()	
+ getDuration()	
+ getTeamMembers()	
+ linkToNote()	
+ addAdvert()	
+ listAdverts()	
+ recordPayment()	
+ getCampaignDetails()	
- getOverheads()	
+ completeCampaign()	

Illustrative Structured English for the calcCosts() operation.

If commissioned then ...
If active then ...
If completed then ...
If paid then ...

© 2010 Bennett, McRobb and Farmer

Behavioural Patterns: State

Potential Solutions : using nested if statements

*How can an object's behaviour change at run-time according to the **state** it is at?*

The state-dependent operations may be implemented using `case` statements or nested `if` statements to specify *alternative behaviour*.

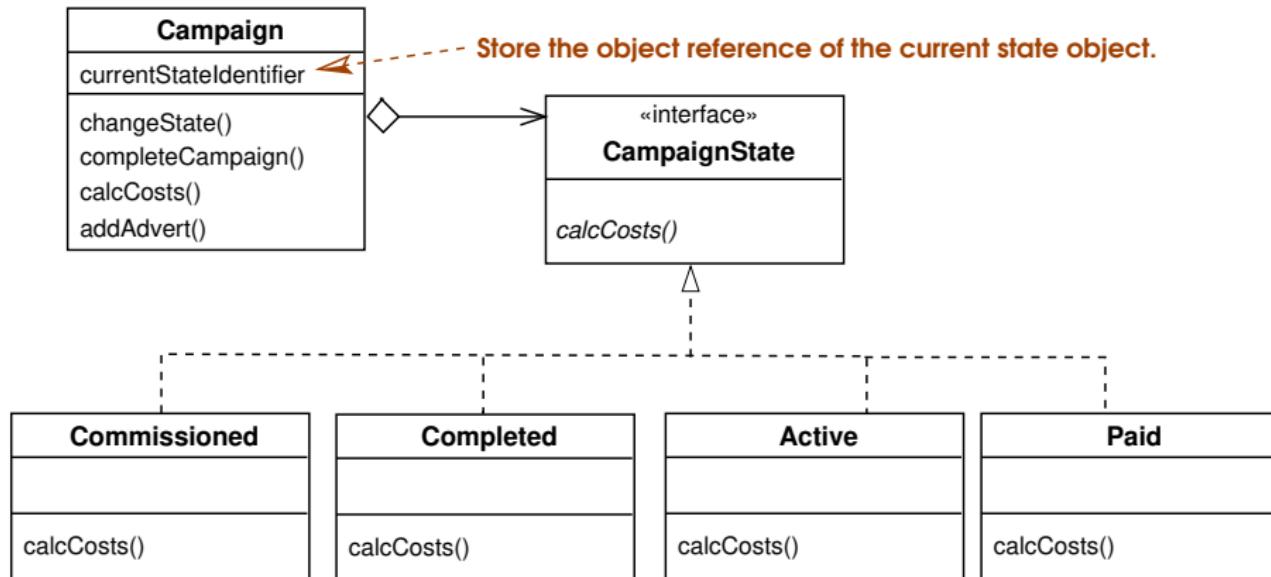
- What is the issue with this approach?
- ▶ The resulting nested `if` statement may become very complex...
 - Especially when an object has many different states.
- ▶ Any addition and/or removal of state(s) will require changes made in the nested `if` statement.

Behavioural Patterns: State

Potential Solutions : using separate components

*How can an object's behaviour change at run-time according to the **state** it is at?*

2. The class may be factored into *separate components*: one for modelling the required behaviours in each of its states.

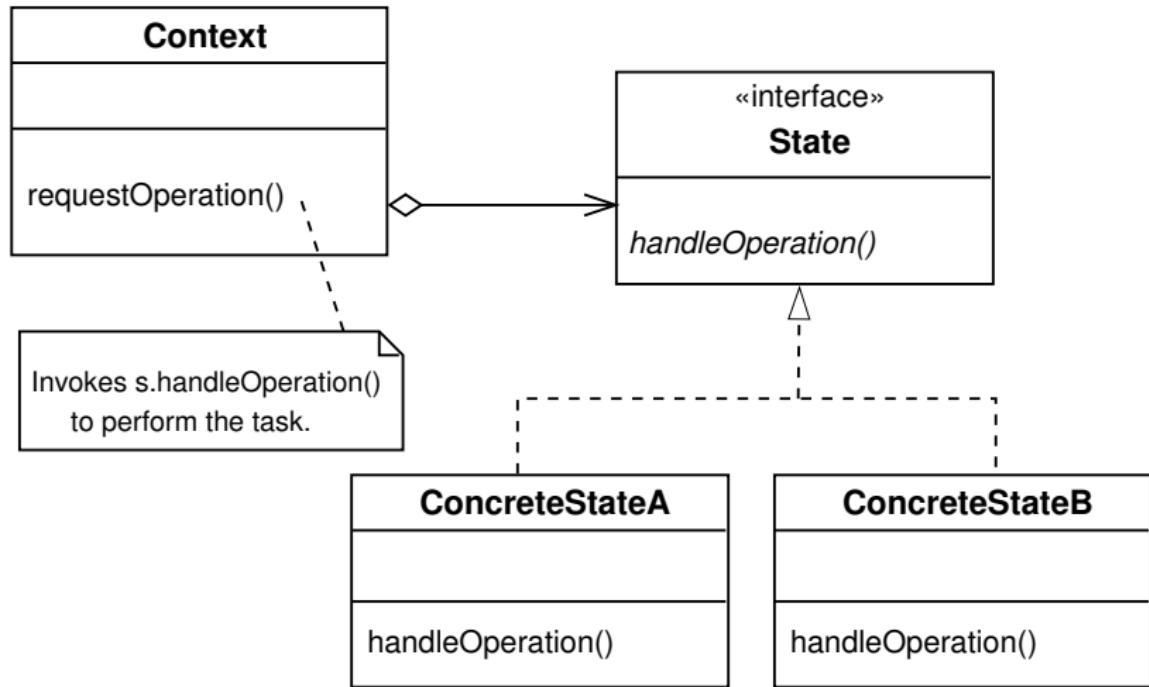


Behavioural Patterns: State Solution

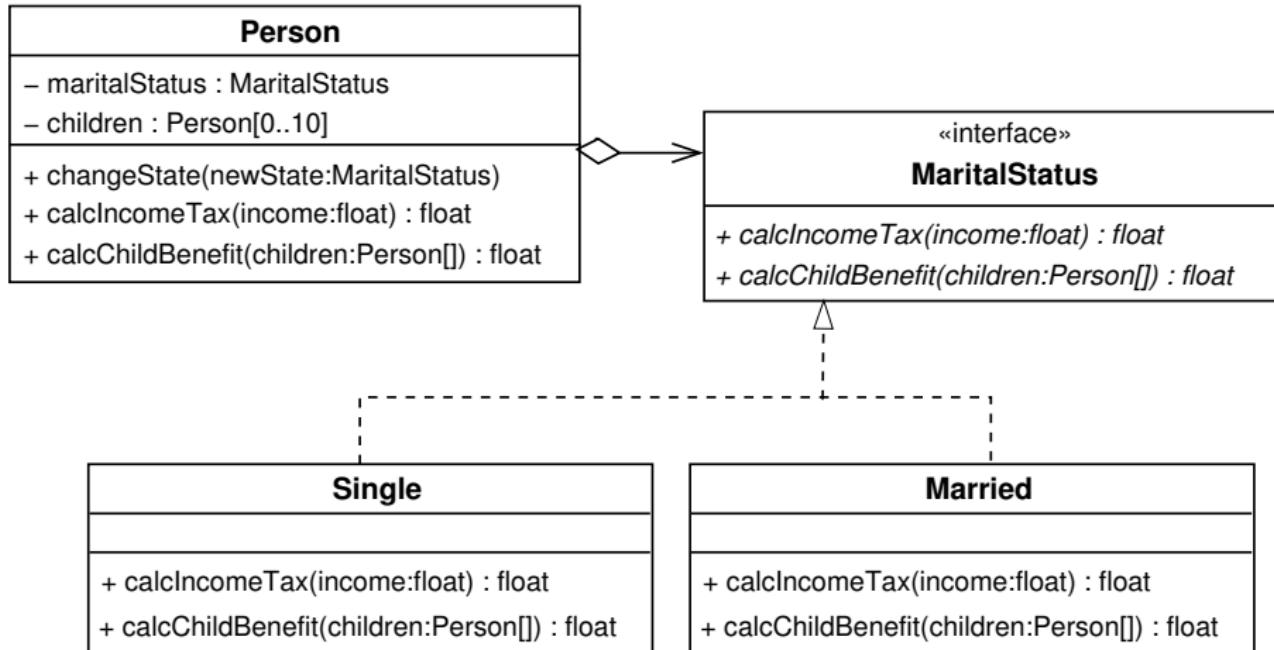
- ▶ ‘An object whose behaviour changes depending on the state that it is in delegates its *state-dependent operations* to other objects which are designed to model state-dependent behaviour’.
- ▶ Each state that the object can be in is modelled by a **concrete class**. These classes implement the same **interface** which specifies state-dependent operations.
- ▶ The object keeps a reference to the current state object, which provides **polymorphic behaviour**.

Behavioural Patterns: State

General Form



Behavioural Patterns: State Application



Application : code example

► Interface MaritalStatus:

```
1 public interface MaritalStatus {  
2     /** Calculate and returns the amount of income tax to be charged */  
3     public abstract float calcIncomeTax(float income);  
4     /** Calculate and returns the amount of child benefit entitled. */  
5     public abstract float calcChildBenefit(Person[] children);  
6  
7 }
```

► Class Single

```
1 public class Single implements MaritalStatus {  
2     public float calcIncomeTax(float income) {  
3         // standard income tax charge...  
4         // method detail omitted  
5     }  
6  
7     public float calcChildBenefit(Person[] children) {  
8         // more child benefit for single parent...  
9         // method detail omitted  
10    }  
11 }
```

Application : code example Person

Class Person

```
1 public class Person
2 {
3     private static final int MAX_CHILDREN = 10;
4
5     private MaritalStatus maritalStatus;
6     private Person[] children;
7     private Person spouse;
8     private float income;
9
10    public Person() {
11        maritalStatus = new Single();
12        children = new Person[MAX_CHILDREN];
13        spouse = null;
14        income = 0.0f;
15    }
16
17    public MaritalStatus getMaritalStatus() {
18        return maritalStatus;
19    }
20    // other fields and methods omitted
21 }
```

Behavioural Patterns: State

Application : code example Person

- To provide state-specific behaviour, the actual calculation of income tax and child benefit entitlement are done by the **state** object:

```
public class Person {  
    private MaritalStatus maritalStatus;  
  
    // Other definitions omitted  
  
    public float calcIncomeTax(float income) {  
        return maritalStatus.calcIncomeTax(income);  
    }  
  
    public float calcChildBenefit(Person[] children) {  
        return maritalStatus.calcChildBenefit(children);  
    }  
}
```

Behavioural Patterns: State

Application : code example Person

- ▶ Over their life, a person's marital status in a software system may need to be changed.
Method `registerSpouse(Person)` invokes method `changeState(Maritalstatus)`.

```
1 public void registerSpouse(Person spouse) {  
2     this.spouse = spouse;  
3     changeState(new Married());  
4 }  
5  
6 public void changeState(MaritalStatus newState) {  
7     maritalStatus = newState;  
8 }
```

From then on, calculation of income tax and child benefit entitlement will be done using operations defined in class `Married`.

Behavioural Patterns: State

Discussion : 1

- ▶ The *state* pattern enables an object to exhibit different behaviour at *run-time* in a flexible manner.
- ▶ **State objects** which provide different behaviour for the same operation may be *created at run-time* and *attached* to the **context object** whenever the circumstance of the modelling entity changes.

Behavioural Pattern: Strategy

Context/Issue

- ▶ Consider a word processing application which supports different ways to align the text in each paragraph.
- ▶ Depending on the type of document, a different alignment strategy would apply.
- ▶ *How to avoid hard-coding all the available alignment strategies in the Composition class which is responsible for laying out the text in a document?*
- ▶ *How to ensure that introducing new alignment strategies in future will require minimal changes to existing classes?*

Behavioural Pattern: Strategy Potential Solutions

- ▶ Hard-code all alignment strategies in the `format` method using a set of **if-then-else** statements:

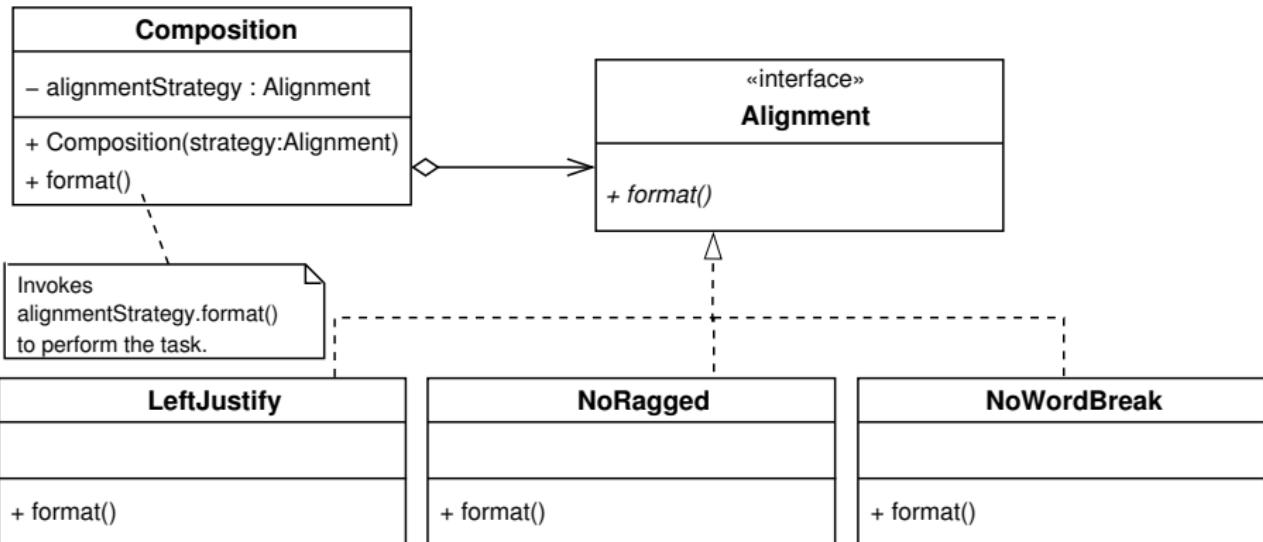
```
1 public void format() {  
2     if (isExamPaper(text)) {  
3         // allow the text to be ragged along the left margin  
4     }  
5     else if (isBook(text)) {  
6         // disallow ragged along both margins  
7     }  
8     else {  
9         // disallow spreading a word across two lines using hyphens  
10    }  
11 }
```

Issue(s) with this approach?

Behavioural Pattern: Strategy

Potential Solutions : 2

- ▶ Define *each* alignment strategy in a *separate class* and make all of them **implement** the same **interface**.
- ▶ Class **Composition** will interact with a suitable alignment strategy through the interface.

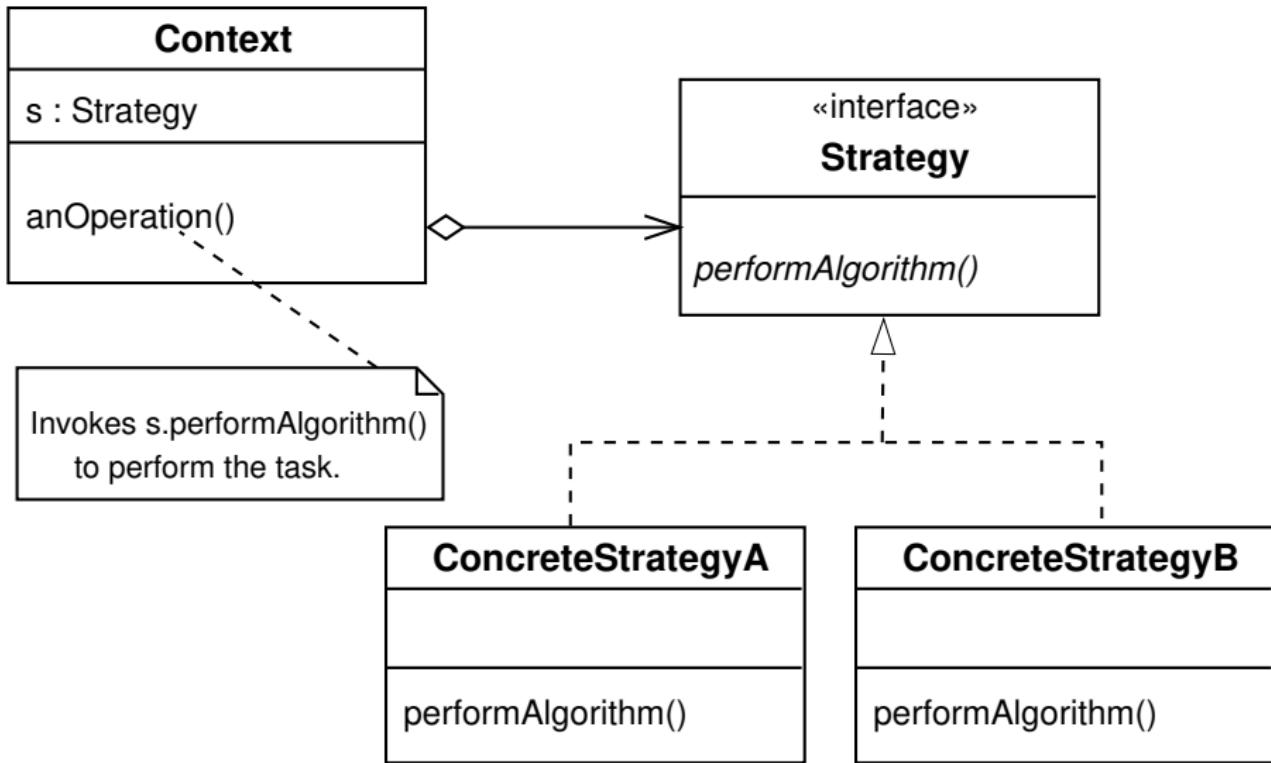


Behavioural Pattern: Strategy Solution

- ▶ Define a *family of algorithms* using an *interface* and a set of *concrete classes*.
- ▶ The **client class** may use any *one* of the algorithms.
- ▶ Existing algorithms may change *without* having any impact on the **client** class.

Behavioural Pattern: Strategy

General form



Behavioural Pattern: Strategy Discussion

- ▶ The *strategy* pattern enables a class to delegate its task to more specialised objects. The **context** class will not need to keep data required by the family of algorithms. The resulting **context** class will be less bloated.
- ▶ In order to perform the required operation, each **concrete strategy** class needs access to the data in the **context** class. One way to achieve this is by passing the reference of the **context** object to the collaborating **concrete strategy** object.

Discussion : code example

► Class IntroductoryOffer:

```
1  public class IntroductoryOffer {  
2  
3      // constructor and other fields and methods omitted  
4  
5      public double applyPricingScheme(Campaign c) {  
6          // apply 10% discount on calculation of all charges  
7      }  
8  }
```

► Class Standard:

```
1  public class Standard {  
2  
3      // constructor and other fields and methods omitted  
4  
5      public double applyPricingScheme(Campaign c) {  
6          // calculate all charges for this campaign  
7      }  
8  }
```

Discussion : code example

► Class Campaign:

```
1  public class Campaign {  
2      private Pricing strategy;  
3  
4      // irrelevant details of the constructor omitted  
5      public Campaign(Pricing strategy, ...) {  
6          this.strategy = strategy;  
7          ...  
8      }  
9  
10     // other fields and methods omitted  
11  
12     public double calculateCost() {  
13         double charge = strategy.applyPricingScheme(this);  
14         ...  
15     }  
16 }
```

Which pricing strategy to use is determined before the campaign come into existence. Class Campaign does not need to know all pricing strategies available.

Discussion : code example

► Class SalesSubsystem:

```
1  public class SalesSubsystem {  
2  
3      // constructor and other fields and methods omitted  
4  
5      public void addCampaign(...) {  
6          ...  
7          // The standard pricing scheme applies.  
8          Campaign c = new Campaign(new Standard());  
9          ...  
10     }  
11 }
```

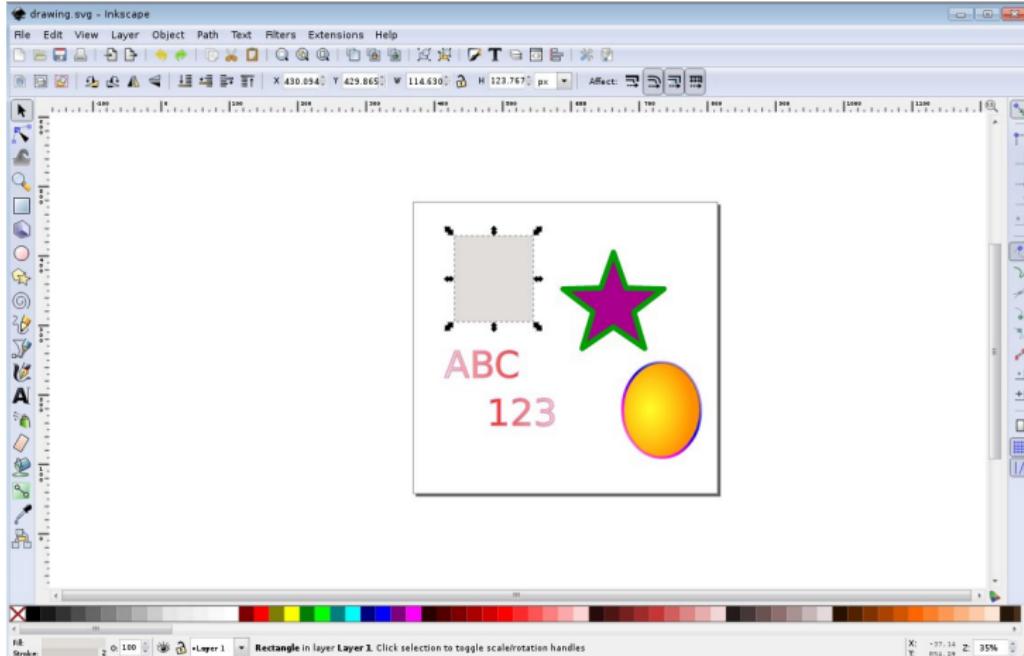
Each pricing strategy can be modelled using the *singleton* pattern.

Difference between State and Strategy?

- ▶ Both use polymorphic classes.
- ▶ Both have a context (e.g., editing or browsing).
- ▶ Both actually apply different strategies (e.g. use different versions of an algorithm).
- ▶ Both are trying to avoid inflexible nested case- or if-statements.
- ▶ Both are actually about doing work in alternative ways, and a ConcreteState actually encapsulates its own group of strategies for executing functions.
- ▶ ‘The StatePattern is a way to vary an object’s behavior dynamically, whereas the StrategyPattern is a way to vary the implementation’

Structural Pattern: Adaptor Context/Issue

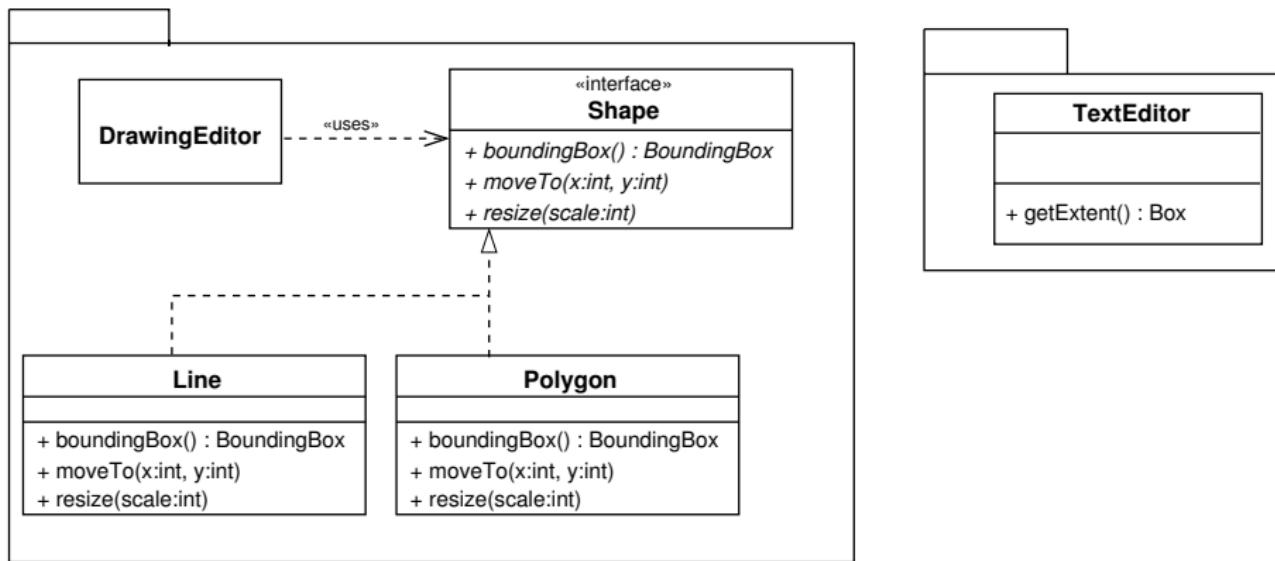
- ▶ Consider a software application (e.g. MyDraw) for users to draw and arrange **graphical elements** (e.g. lines, polygons, text, ...) into a diagram.



Structural Pattern: Adaptor Context/Issue

- ▶ We define an **abstract class** (e.g. `Shape`) for modelling a **generic** graphical element. Subclasses of `Shape` would model `Line`, `Polygon`, `Circle`, etc.
We also need a `Text` subclass for displaying and editing text.
- ▶ There is a `TextEditor` **framework** class which addresses the requirements of our intended `Text` class.
- ▶ However, `TextEditor` does not have the same interface as the **abstract class** `Shape` which `MyDraw` expects to work with.

Structural Pattern: Adaptor Context/Issue



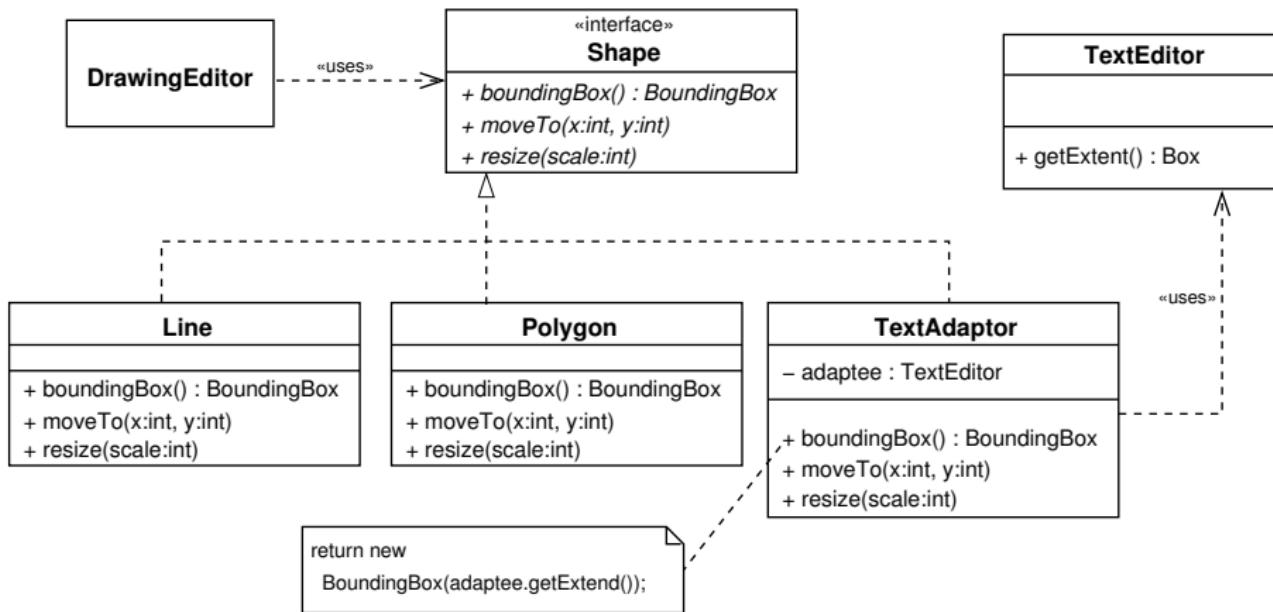
How to enable an existing class to work with classes with a different and incompatible interface?

Structural Pattern: Adaptor Solution

Gamma et al. (1995, pp.157–160):

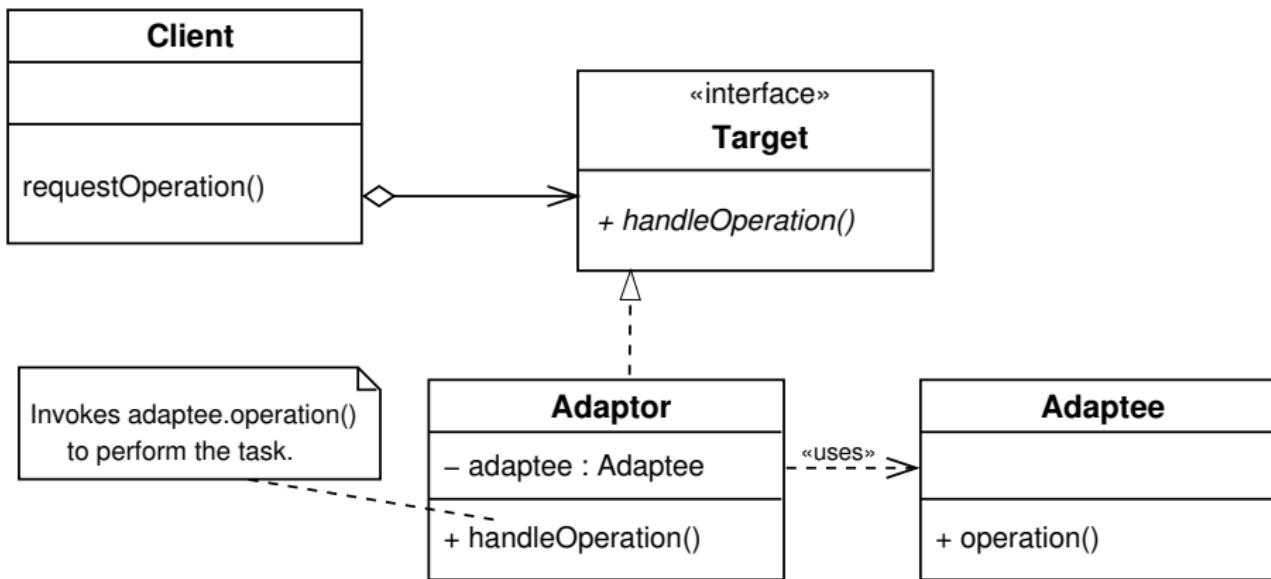
- ▶ Using an **Adapter** class, convert the interface of a class into another interface which clients expect.
- ▶ *Clients call operations on an **Adaptor instance**.*
- ▶ The **adapter** then invokes the respective **Adaptee** operations to complete the task.
- ▶ The **Adapter** class will also include **concrete implementation** of other methods which are defined by the implementing **interface** but not supported by the **Adaptee** class.

Structural Pattern: Adaptor Solution



Structural Pattern: Adaptor

General form



Structural Pattern: Adaptor

Examples from Java API

- ▶ Examples of the **adaptor** design pattern in the Java API include:
 - ▶ `java.util.Arrays.asList`
 - ⇒ Method `asList` **adapts** an `array` to the collection type `list`.
- ▶ An **adaptor** is particularly useful when you want to use an existing class, but its interface does not match the one required by your application.
- ▶ See Larman (2005, pp.436–440) for another example.

Applying Patterns

Before

Before using a pattern to resolve the problem, consider:

- ▶ Is there a simpler solution?
Patterns should not be used just for the sake of it.
- ▶ Is the context of the pattern consistent with that of the problem?
- ▶ Are the consequences of using the pattern acceptable?
- ▶ Are constraints imposed by the software environment that would conflict with the use of the pattern?

Applying Patterns

After selecting a suitable pattern...

Steps to follow in applying the selected pattern:

1. Read the pattern to get a complete overview.
2. Study the **Structure**, **Participants** and **Collaborations** of the pattern in detail.
3. Examine **Sample Code** to see an example of the pattern in use.
4. Choose names for the pattern's participants (i.e. classes) that are meaningful for your application.
5. Define the classes, with operations that perform the responsibilities and collaborations in the pattern.

References

- ▶ Braude, E.J. and Bernstein, M.E., *Software Engineering: Modern Approaches*, 2nd ed, Hoboken, NJ: Wiley, 2011.
- ▶ Gamma, E., Helm, R., Johnson, R. and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, MA: Addison-Wesley, 1995.
- ▶ Larman, C., *Applying UML and patterns*, 3rd ed, Upper Saddle River, NJ: Prentice Hall, 2005.

For detail, see:

Chapter 17 of Braude & Bernstein (2011),

Larman (2005) Chapter 26 and

Gamma et al (1995).

Learning Outcomes. You should now be able to

- ▶ describe what is meant by design pattern.
- ▶ state examples of design patterns.
- ▶ identify the application of design patterns in some Java packages.
- ▶ apply some basic design patterns in software development.
- ▶ state the benefits and difficulties that may arise when applying patterns in software development.

Unit 12 Implementation

Unit Outcomes. Here you will learn

- ▶ about the tasks involved in implementation
- ▶ techniques in writing maintainable code
- ▶ what is meant by defensive programming and how to enforce intention
- ▶ about tools used in software implementation

Further Reading: Bennett et al. Ch19 and Braude & Bernstein Ch22

based on Bennett, McRobb & Farmer (2010) and Braude & Bernstein (2011)

Implementation

What does it involve?

Two different views:

- ▶ Bennett et al. (2010, p.559) wrote:

"Implementation is concerned with the process of building the new system. This involves writing program code, developing database tables, testing the new system, setting it up with data, possibly transferred from an old system, training users and eventually switching over to the new system."

- ▶ Braude & Bernstein (2011, p.35), however, stated:

The implementation phase "consists of programming, which is the translation of the software design developed in the (design) phase to a programming language. It also involves integration, the assembly of the software parts."

Implementation

What does it involve?

- ▶ One thing in common is that **implementation** involves constructing a software system based on a design: this involves *writing program code*.
- ▶ To facilitate the coding process, one needs to consider:
 - ▶ Which programming *languages* to use?
 - ▶ Which *software tools* will be used to support the development process?
 - ▶ How to ensure that the code, though written by different programmers, will be *easy to read, understand and maintain*?
 - ▶ How to effectively *manage changes to programs* throughout the development process?

Language choice (1)

The choice of programming language(s) mainly depends on the nature of the required software system.

- ▶ e.g. *JavaScript, CSS, HTML* for *client-side* Web applications.
- ▶ For *server-side* scripting, could be *PHP, Java EE, ASP.NET, Ruby, R, python*
- ▶ Lots of simultaneous, lightweight communications? maybe *Node.js with websockets*
- ▶ Where a client prefers to use Microsoft products only, the implementation language is likely to be *C#*.

Language choice (2)

- ▶ If the tasks require a substantial amount of processing on *tree* data structures, a *functional programming language* such as *Lisp*, *Haskell* or possibly *Julia* is likely to be a good choice.
- ▶ If a system is expected to perform a lot of *low-level system operations* and *efficiency* is paramount, *C* may be a preferred language choice.
- ▶ If the required software system is expected to run on *different platforms* a language like *Java* or *Scala* would be appropriate.

Programming Conventions

Writing Classes

- ▶ Class names should be *meaningful*, e.g. in the Agate example, CampaignStaff would be a better class name than Staff.
- ▶ Avoid using *ambiguous* or *cryptic* names such as Record or c1.
- ▶ The purpose of each class should be *clearly* and *concisely* stated in the documentation comment.
 - ➡ This view is not universally agreed. Some software developers adopt a “comment-free” practice, and opt to achieve clarity by structuring the source code and naming identifiers, methods, variables, etc. in a way that make the code self-explanatory.

Programming Conventions

Documenting Methods

- ▶ The *purpose* of each method should be clearly stated in code comments, e.g.:

```
/** Tests whether this date is after the specified date.  
 * @param when the specified date  
 * @return true if this date is after when, false otherwise  
 */  
public boolean after(Date when) // detail omitted.
```

- ▶ To facilitate *maintenance* and *reuse*, it is helpful to specify the following information for methods, especially for the *non-trivial* ones:
 - ▶ intent
 - ▶ return
 - ▶ exceptions
 - ▶ known issues
 - ▶ preconditions
 - ▶ post-conditions
 - ▶ invariant

Programming Conventions

Documenting Methods (Braud & Bernstein, 2011)

- ▶ **Intent:**

- ▶ An *informal* statement of what the method is intended to do.

- ▶ **Return:**

- ▶ The value which the method returns.

- ▶ **Known issues:**

- ▶ Honest statement of what has to be done, defects that have been repaired, etc.

Programming Conventions

Documenting Methods (Braud & Bernstein, 2011)

► **Exceptions:**

- ▶ Capture the situations when an *abnormality* occurred during the *execution* of the method.
- ▶ Such situations are often caused by the **preconditions** not having been met.

Programming Conventions

Documenting Methods (Braud & Bernstein, 2011)

► **Preconditions:**

- What must be true *when* a method is invoked.
- Conditions on non-local variables, including input parameters, that the method assumes, e.g.:

```
private int price;  
// other detail omitted.  
/* Precondition: reduction > 0 */  
public void reducePrice(int reduction) {  
    price -= reduction;  
}
```

- Verification of these conditions is not guaranteed by the method as it is assumed to be done by the caller.

Programming Conventions

Documenting Methods (Braud & Bernstein, 2011)

► Post-conditions:

- What must be true *after* a method completes successfully.
- Specify the effect of the method.
- Specify the value of non-local variables after execution.
- Notation: x' denotes the value of variable x *after* execution, e.g.:

```
private int price;  
  
    // other detail omitted.  
  
    /* Post-condition: price' = price - reduction */  
    public void reducePrice(int reduction) {  
        price -= reduction;  
    }
```

Programming Conventions

Class invariants

- ▶ *Class invariants* specify *constraints* on **attributes** and their relationships. e.g.:

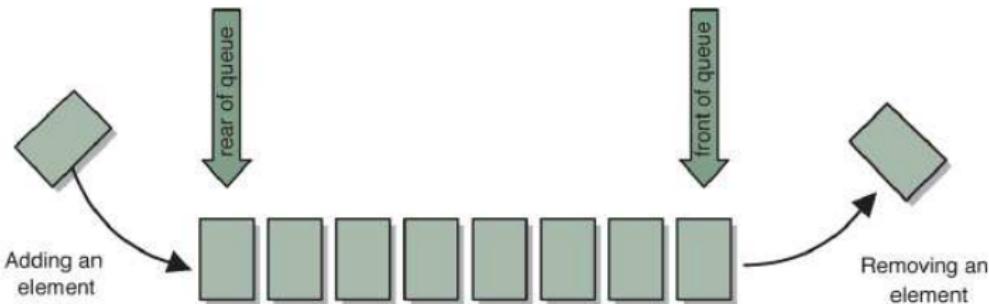
```
public class Time {  
    private int hour;    // hour >= 0 && hour < 24  
    private int minute; // minute >= 0 && minute < 60  
    private int second; // second >= 0 && second < 60  
    // other detail omitted.  
}
```

- ▶ Even though the value of each attribute in an object might change at runtime, its values must conform to the *class invariants*.

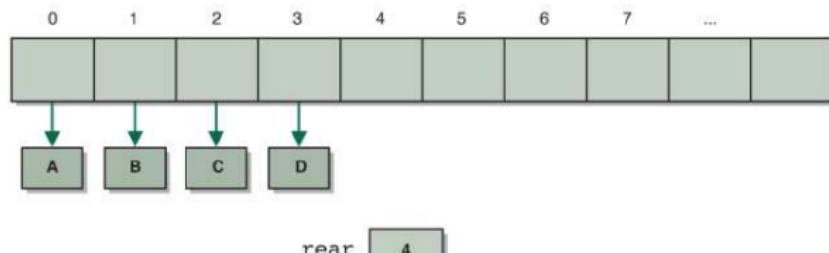
Programming Conventions

Class invariants : an exercise

Consider the following data structure:



BoundedArrayQueue is a queue data structure modelled by an array. The capacity of the queue is fixed.



Programming Conventions

Class invariants : an exercise

Write the **invariants** for class `BoundedArrayQueue`:

```
public class BoundedArrayQueue<T> {
    /* Class invariant: ... */
    private T[] contents;
    private int rear; // the index of the next vacant cell
    private int size;

    public BoundedArrayQueue(int capacity) // detail omitted.
    public boolean enqueue(T element) // detail omitted.
    public T dequeue() // detail omitted.
}
```

Programming Conventions

Class invariants : an exercise

```
public class BoundedArrayQueue<T> {
    /* Class invariant: ... */
    private T[] contents;
    private int rear; // the index of the next vacant cell
    private int size;

    public BoundedArrayQueue(int capacity) // detail omitted.
    public boolean enqueue(T element)      // detail omitted.
    public T dequeue()                   // detail omitted.
}
```

```
/* Class invariant:
 *   size >= 0 && size <= contents.length
 *   rear == size
 */
```

Java Assertions

- ▶ **Preconditions**, **post-conditions** and **invariants** can be specified in a Java program as **assertions**.
- ▶ *Assertion* in Java provides a convenient tool for run-time checking.
 - ⇒ as well as a means for documentation
- ▶ An `assert` statement typically:
 - ▶ performs a *check*, and
 - ▶ displays a *message* when the check fails.
- ▶ An `assert` statement may also:
 - ▶ throw an exception, and
 - ▶ include file and line info.

Java Assertions

- ▶ By default, assertions are *ignored* by the JVM and they are used mainly for *debugging*.
- ▶ To *switch on* assertion checks, a Java program needs to be run with the option `-ea` (i.e. `e`nable `a`ssertions), e.g.:

```
java -ea ArrayRemove
```

where `ArrayRemove` is a compilation unit that includes a `main` method.

N.B. Do not confuse `assert` methods in *JUnit* with `assert` statements in Java. They serve different purposes.

Java Assertions

Using assertions in Java : Example

```
public static String removeElement(String[] array, int pos) {  
    // check a precondition:  
    assert 0 <= pos && pos < array.length :  
        "removeElement: precondition failed";  
    // remember the original array for post-condition checks:  
    String[] originalArray = array.clone();  
    String result = array[pos];           // the result  
    // shift all elements after pos one position left:  
    for (int i = pos; i < array.length - 1; i++) {  
        array[i] = array[i + 1];  
    }  
    array[i] = null; // the last element becomes null  
    // check some post-conditions:  
    assert array[array.length - 1] == null:  
        "removeElement: post-condition failed";  
    assert pos + 1 == array.length  
        || array[pos].equals(originalArray[pos + 1]):  
        "removeElement: post-condition failed";  
    return result;  
}
```

Coding Standards

Object-oriented standard

- ▶ Naming standards should be agreed early in a project.
- ▶ A typical *object-oriented standard*:
 - ▶ Classes with capital letters: Campaign
 - ▶ Attributes and operations with initial lower case letters:
`title`, `recordPayment()`
 - ▶ Words are concatenated together with capital letters to show where they are joined, e.g.:
 - ▶ InternationalCampaign,
 - ▶ campaignFinishDate,
 - ▶ getNotes()

Coding Standards

Hungarian Notation

- ▶ The *Hungarian notation* is typically used in *C* or *C++*.
- ▶ Identifiers are prefixed by an abbreviation to show the *type* of the variable:
 - ▶ `b` for boolean: `bOrderClosed`
 - ▶ `i` for integer: `iOrderLineNumber`
 - ▶ `btn` for button: `btnCloseOrder`

Coding Standards

Underscore convention

- ▶ Another convention commonly used for *column names* in *databases* uses *underscores* to separate parts of a name instead of capital letters, e.g.:

Order_Closed

... or, in Postgres...

order_closed

- This convention makes it easier to replace the underscores with spaces to produce meaningful column *headings* in reports.

Documentation Standard

Document code

- ▶ The one who wrote a piece of code is unlikely to be the one having to maintain it in future. Hence, it is important that code is *readable* to facilitate maintenance.
- ▶ Having *good documentation* is *important* to facilitate maintainability and reusability of any written code.
- ▶ Java has well-defined *documentation standards* which should be adhered to if coding in *Java* (Javadoc).
- ▶ Use *XML* tags if coding in *C#*.
- ▶ Many IDEs include features for generating *Javadoc* from comments automatically.

How to Write Readable Java Programs?

Motivating example (1)

- ▶ A piece of rather *unreadable* code:

```
// another method to help me
private static void raa (char[] a)
{ final int s = a.length; for (int i = 0; i < s/2; i++)
char c = a[i]; a[i] = a[s-i-1];
a[s-i-1] = c;
}
```

How to Write Readable Java Programs?

Motivating example (2)

- ▶ Better...

```
/**  
 * reverse an array of characters in its own space  
 * @param array  
 */  
private static void reverseArray (char [ ] array)  
{  
    final int size = array.length;  
    for (int i = 0; i < size/2; i++) // first half of array  
    {  
        // swap i'th and mirrored (size - i - 1)'st element  
        char c = array[i]; // temporary storage  
        array[i] = array[size - i - 1];  
        array[size - i - 1] = c;  
    }  
}
```

- ➡ but still can be improved...

How to Write Readable Java Programs?

Motivating example (3)

- ▶ even clearer:

```
private static void reverseArray (char[] array)
{
    int index = 0; // start with the first element
    int mirrorIndex = array.length - 1; // and the last
    while ( index < mirrorIndex )
    {
        // swap index and mirrorIndex elements
        char c = array[index];
        array[index] = array[mirrorIndex];
        array[mirrorIndex] = c;
        // advance counters
        index++;
        mirrorIndex--;
    }
}
```

How to Write Readable Java Programs?

Naming conventions for...

- ▶ *packages*: all lower case
 - ▶ internal use: prefix with name of product, e.g.: `cwk1.shapes`
 - ▶ for distributing: prefix with domain name, e.g.:
`uk.ac.aston.cs`

The word in between each fullstop corresponds to the name of a *folder* in the file system.

- ▶ *reference types* (i.e. class/interface names): start with a capital, then mixed case.
- ▶ *classes*: use a noun describing an instance, e.g.: `MyShape`.
- ▶ *interfaces*: either a noun, e.g. `Observer`, or an adjective, e.g. `Cloneable`.

How to Write Readable Java Programs?

Naming conventions for...

- ▶ *methods*: start with lower case, than mixed case
 - ▶ Active methods (i.e. **mutator methods**) usually start with verb, e.g. `changeSize()`
 - ▶ Getter methods (i.e. **accessor methods**), which instigating no change, are named by return value, e.g. `length()`, or with a '**get**', e.g. `getLength()`.
- ▶ *fields and constants*
 - ▶ **Static final** fields (i.e. **named constants**): all capitals, e.g. `EQUALS_TOLERANCE`, `INITIAL_VALUE`.
 - ▶ All other fields use the same capitalisation as method names, e.g. `size`, `isVisible`.

How to Write Readable Java Programs?

Naming conventions for...

- ▶ *parameters*: same capitalisation as method names,
e.g. newSize
- ▶ Parameters in **public** methods appear in the documentation.
- ▶ Choose fitting, yet succinct names for parameters.
- ▶ Parameters having the same meaning should have the same name, even when they appear in different methods,
e.g.:

```
distance(Location anotherLoc)
```

```
almostEqual(Location anotherLoc)
```

How to Write Readable Java Programs?

Naming conventions for...

- ▶ *local variables*: same capitalisation as method names, e.g.:
count
- ▶ Always use *meaningful* names good for achieving readability.
- ▶ Except for routine cases, e.g. `for(int i=0; ...; i++)`

How to Write Readable Java Programs?

Layout conventions

- ▶ Matching braces must line up vertically or horizontally
- ▶ every method except `main` should be preceded by a *doc comment*.
- ▶ A method definition 'should not be longer than 30 lines'.
- ▶ `switch` statement should **not** be used, except with `enum` values.
 - some examples here:
<https://www.baeldung.com/java-switch>
- ▶ All keywords and operators should be surrounded by a space:

```
if (x == 0) y = 0; // good  
if(x==0) y=0;      // bad
```

How to Write Readable Java Programs?

Layout conventions

- ▶ No “*magic numbers*” should be used (i.e. direct usage of numbers in the code)
- ▶ Some exceptions for numbers with generic meanings i.e. `(-1, 0, 1, 2)`
BUT...
- ▶ Clearly-named constants or methods are still preferable
- ▶ e.g. `number % 2 == 0`
- ▶ ...could be replaced with an `isEven` function

How to Write Readable Java Programs?

Layout conventions

Exercise: Spot the magic number(s)...

```
public static int timeToPension(int ageNow, int currentYear)
{
    int statePensionAge = 65; // initial assumption
    if (currentYear + 65 - ageNow >= 2024) {
        statePensionAge = 66;
    }
    if (currentYear + 66 - ageNow >= 2034) {
        statePensionAge = 67;
    }
    if (currentYear + 67 - ageNow >= 2044) {
        statePensionAge = 68;
    }
    return statePensionAge - ageNow;
}
```

How to Write Readable Java Programs?

Layout conventions

```
final int SPA_66_YEAR = 2024;
final int SPA_67_YEAR = 2034;
final int SPA_68_YEAR = 2044;

public static int timeToPension(int ageNow, int currentYear) {
    int statePensionAge = 65; // initial assumption
    if (currentYear + 65 - ageNow >= SPA_66_YEAR) {
        statePensionAge = 66;
    }
    if (currentYear + 66 - ageNow >= SPA_67_YEAR) {
        statePensionAge = 67;
    }
    if (currentYear + 67 - ageNow >= SPA_68_YEAR) {
        statePensionAge = 68;
    }
    return statePensionAge - ageNow;
}
```

How to Write Readable Java Programs?

Documentation comments (aka doc comments)

- ▶ start with `/**` and end with `*/`
- ▶ are used to automatically generate documentation of types, methods and fields (e.g. in *Eclipse*, *NetBeans*, *BlueJ*)
- ▶ may contain simple *HTML* tags (e.g. for bold style, bullet points)

How to Write Readable Java Programs?

Documentation comments (aka doc comments)

- ▶ may contain a number of special **tags**, e.g.:
 - ▶ @author
 - ▶ @version
 - ▶ @see hyperlink to documentation of another member, class
 - ▶ @param every method parameter should have one
 - ▶ @return describes the return value of a method
 - ▶ @throws describes potential exception propagation

Defensive Programming

What is it? (Braude & Bernstein, 2011)

- ▶ *Defensive Programming* is a programming technique which seek to minimise program errors by *anticipating* potential errors and *implementing* code to handle them.
- ▶ Techniques for defensive programming include:
 - ▶ various error handling techniques
 - ▶ exception handling
 - ▶ buffer overflow prevention
 - ▶ enforcing intentions

Defensive Programming

Error handling techniques

- ▶ *Wait for a legal data value*: typically used when getting data from an external source, e.g. user interface
- ▶ *Set a default value*: typically used to ensure the continuity in execution of a (critical) process
- ▶ *Use the previous result*: typically used when a constant stream of inputs is expected at a regular interval
- ▶ *Log error*: store error info for later use or analysis
- ▶ *Throw an exception*
- ▶ *Abort*: typically used in operations where illegal data may lead to a fatal consequence, hence the system must be aborted and reset.

Defensive Programming

Buffer overflow prevention

- ▶ Some languages (e.g. **C** and **C++**) permit data to be written to memory even if the data require more space than what was declared in the code, *so long as* the required space does not exceed the space allocated to the program (e.g. through the **malloc** function).
- ▶ This may lead to *overwriting* existing data kept in the memory.
- ▶ Such overwriting, when exploited, may lead to a *security breach*.
- ▶ **Buffer overflow prevention** attempts to prevent buffer overflow by checking the size of variables.

Defensive Programming

Enforce intention

- ▶ Use suitable programming constructs to enforce the intended design or use of a piece of code.
- ▶ Use keywords such as `final` and `abstract` to enforce coding intentions, e.g.:

```
public final class String {...}  
public abstract class AbstractList {...}
```

- ▶ Make constants, variables and classes as *local* as possible, e.g.:

```
for (int i = pos; i < array.length - 1; i++) {...}
```

rather than:

```
int i;  
for (i = pos; i < array.length - 1; i++) {...}
```

Defensive Programming

Enforce intention

- ▶ Use the **singleton** design pattern if only one instance is expected from a class.
- ▶ Define attributes and operations as **private** if they are not intended to be accessible by other classes.
- ▶ If a class is expected to be **extended** by other classes, make attributes **protected** and define **accessor methods** for accessing their values.
- ▶ If a method is not expected to be used by classes in another **package**, do not use the **public** keyword to specify its **visibility**.

Defensive Programming

Enforce intention

- ▶ Use *enumerate type* to enable a variable to be assigned with a predefined set of constants, e.g.:

Rather than:

```
int tShirtSize = 1;
```

we define an *enum* type in Java to model permissible sizes for a T-shirt:

```
public enum Size {  
    XS, S, M, L, XL, XXL  
}
```

```
Size tShirtSize = XS;
```

Defensive Programming

Enforce intention

- ▶ Consider introducing *new classes* to encapsulate legal parameter values so as to prevent bad data, e.g.:

Instead of:

```
determineRoadTax(String vehicle)
```

we use:

```
determineRoadTax(Vehicle vehicle)
```

Class **Vehicle** might have various *factory methods* to create the permissible types of vehicles, e.g.:

```
Vehicle createCar() {...}  
Vehicle createLorry() {...}  
Vehicle createMotorbike() {...}
```

Software Implementation Tools

Modelling Tools

- ▶ Many interactive development environments (IDEs) *support UML*.
 - ➡ However, the notations used might not always strictly follow the current standard.
- ▶ Some modelling tools support *automatic generation* of program code (in *Java*, *C++* and *VB*) from the models, e.g. *Visual Paradigm*.
- ▶ Some also support *reverse engineering* of code (i.e. generating a design model from existing code).
- ▶ Some of them map classes to a *relational database*.

Software Implementation Tools

IDEs (Integrated Development Environments)

- ▶ Manage files in a project and the dependencies among them.
- ▶ Link to **configuration management tools**.
- ▶ Use compilers to build the project, *only* recompiling what has changed.
- ▶ Provide various facilities such as *debugging, refactoring, auto-layout, documentation generation*, etc.
- ▶ May include a *visual editor* for GUI building.
- ▶ Can be configured to link in third party tools.

Software Implementation Tools

Configuration Management Tools

- ▶ Include elements of **version control** - to *compare* source code so as to assist in resolving potential **conflicts**.
 - ⇒ Though configuration management is more than just version control.
- ▶ Maintain a record of *file versions* and the *changes* from one version to the next.
- ▶ Record all the versions of software and tools that are required to produce a repeatable software **build**.
- ▶ **Docker** provides many of these configuration management functions.

Software Implementation Tools

DBMS (Database Management Systems)

A DBMS typically comprises:

- ▶ Server system
- ▶ Client software (administration interfaces, ODBC and JDBC drivers)
- ▶ Tools to manage the database and carry out performance tuning
- ▶ Tools to make compiled classes persistent so they can be used with object DBMS
- ▶ Large DBMS, such as [Oracle](#), come with many tools, even their own application server

Software Implementation Tools

Application Containers

- ▶ Web containers, e.g. *Tomcat* (Java), *Flask* (python)
 - ▶ Run servlets and small-scale applications.
- ▶ Application servers, e.g. *GlassFish*, *WebSphere*, *nginx*, *Apache Web Server*
 - ▶ Provide a framework in which to run large-scale, enterprise applications.

Software Implementation Tools

Installation /Deployment Tools

- ▶ e.g. [GitLab CI/CD](#), [AWS CodeDeploy](#), [Jenkins...](#)
- ▶ Automate extraction of files from an archive and setting up of configuration files and registry entries.
- ▶ Some maintain information about dependencies on other pieces of software and will install all necessary packages
- ▶ Uninstall software, removing files, directories and registry entries.

References

- ▶ Bennett, S., McRobb, S. and Farmer, R., *Object-Oriented Systems Analysis and Design Using UML*, 4th ed, Maidenhead: McGraw-Hill, 2010.
- ▶ Braude, E.J. and Bernstein, M.E., *Software Engineering: Modern Approaches*, 2nd ed, Hoboken, NJ: Wiley, 2011.

For detail, see:

Chapter 19 of Bennett et al. (2010),

Chapter 22, Section 6.2.4 and Section 6.4 of Braude & Bernstein (2011).

Learning Outcomes

Learning Outcomes. You should now be able to

- ▶ state the tasks involved in implementation
- ▶ write class invariants, preconditions and post-conditions
- ▶ write readable and easy-to-maintain Java code
- ▶ state what is meant by defensive programming
- ▶ recognise when to apply various error handling techniques
- ▶ use appropriate Java language constructs to enforce intention
- ▶ identify appropriate tools for supporting various implementation tasks

Unit 12 Refactoring

Unit Outcomes. Here you will learn

- ▶ what is meant by refactoring
- ▶ various kinds of refactorings
- ▶ how to use `Eclipse` to perform some refactorings

Further Reading: Braude & Bernstein Ch24

Some Tips on Software Development

Fowler (1999)

- ▶ *"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."*
- ▶ *"When you find you have to add a feature to a program, and the program's code is not structured in a convenient way to add the feature, first refactor the program to make it easy to add the feature, then add the feature."*
- ▶ *"Refactoring changes the programs in small steps. If you make a mistake, it is easy to find the bug."*
- ▶ *"Before you start refactoring, check that you have a solid suite of tests. These tests must be self-checking."*

▶ Fowler (1999)

What is refactoring?

- ▶ “*Refactoring is a process of altering source code so as to leave its existing functionality unchanged.*”

Braude & Bernstein (2011, p.601)

- ▶ *Refactoring* refers to the process of restructuring source code in order to improve its **readability**, **maintainability** and **extendability** without altering any *observable* behaviour of the software system.
- ▶ Though a key reason for refactoring may be to improve the source code’s ability to cope with new requirements, i.e. in preparation for *system extension*, refactoring itself does *not*, and should not, lead to any change in the *functional requirements* which the source code is to address.
 - ➡ This allows the same suite of automated tests to be used.

What is refactoring?

Examples : Renaming

Renaming

- ▶ Renames an element and (if enabled) corrects all references to the elements (also in other files).
- ▶ An element may be a:
 - ▶ method,
 - ▶ type parameter,
 - ▶ method parameter,
 - ▶ enum constant,
 - ▶ field,
 - ▶ compilation unit,
 - ▶ local variable,
 - ▶ package
 - ▶ type,
- ▶ Shift + Alt + R in Eclipse

What is refactoring?

Examples : Promoting attribute to class

Promoting attribute to class

- ▶ Create a new class for modelling an attribute.
- ▶ Typically, the promoted attribute was defined as a **primitive** or **String** type previously.
- ▶ The new class enables further information to be modelled.

before:

```
public class Customer {  
    private String firstName;  
    private String surname;  
    private String address;  
  
    // other field and method  
    // definitions omitted  
}
```

after:

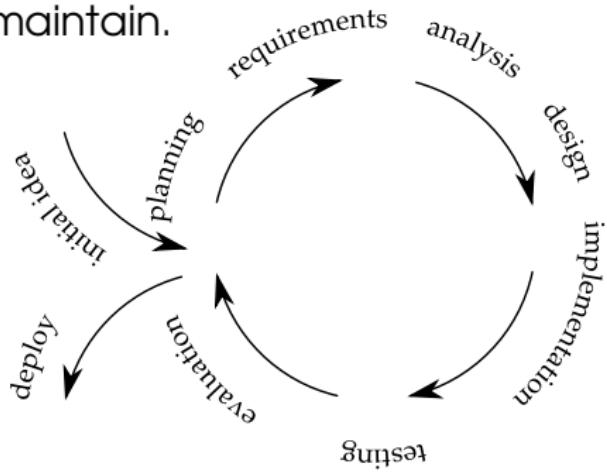
```
public class Customer {  
    private String firstName;  
    private String surname;  
    private Address address;  
  
    // other field and method  
    // definitions omitted  
}
```

Why is there the need for refactoring?

- ▶ Refactoring is *particularly relevant* in **Agile development**.
- ▶ In **Agile** project life cycle, the typical phases in a **waterfall** model are performed *repeatedly* in cycles, i.e. requirements capture, design, implementation, testing, requirements capture, design, implementation, testing, etc.
- ▶ In *each* cycle, new code is written and *added* to the existing **code base**.
- ▶ At each cycle, the focus is to develop code that *addresses the current requirements*.

Why is there the need for refactoring?

- ▶ At the end of each cycle, the **code base** needs to be “cleaned up” in order to prevent it from becoming too messy and difficult to maintain.



- ▶ Refactoring facilitates **maintenance** and **extension** of the code base.

Big Refactorings

- ▶ This kind of refactoring changes the *overall system design* in terms of the number of classes involved.
- ▶ We will consider four kinds of *big refactorings*:
 - ▶ Extract hierarchy
 - ▶ Tease apart inheritance
 - ▶ Convert procedural design to objects
 - ▶ Separate domain from presentation

Big Refactorings

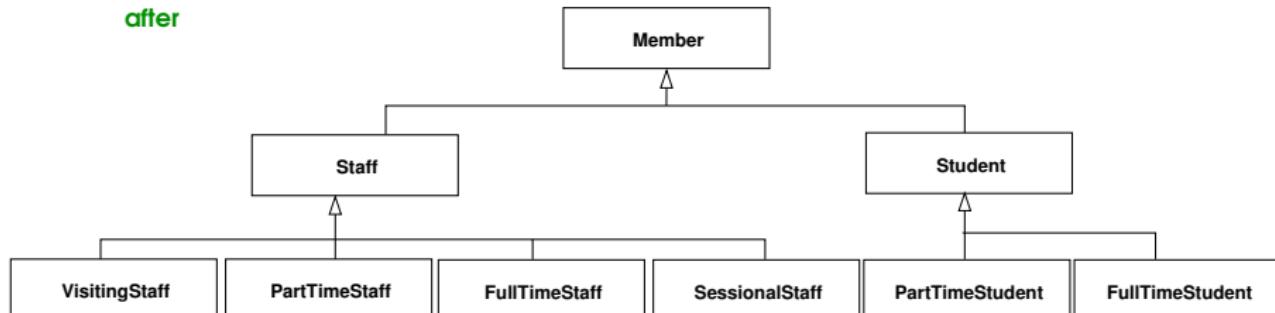
1. Extract hierarchy

- Refine the current class design by introducing a *new class hierarchy* or extending an existing class hierarchy.

before



after



Braude & Bernstein (2011, Figure 24.7)

Big Refactorings

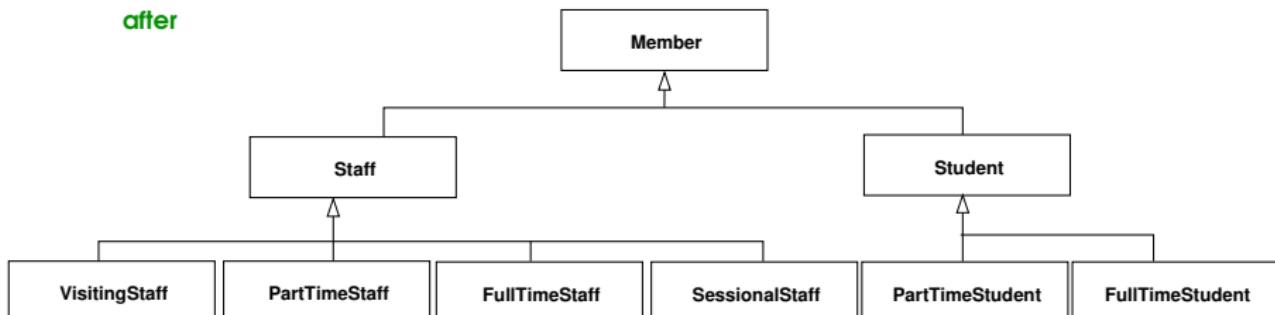
Extract hierarchy - a question

- ▶ What sorts of problems might have led to this refactoring?

before



after

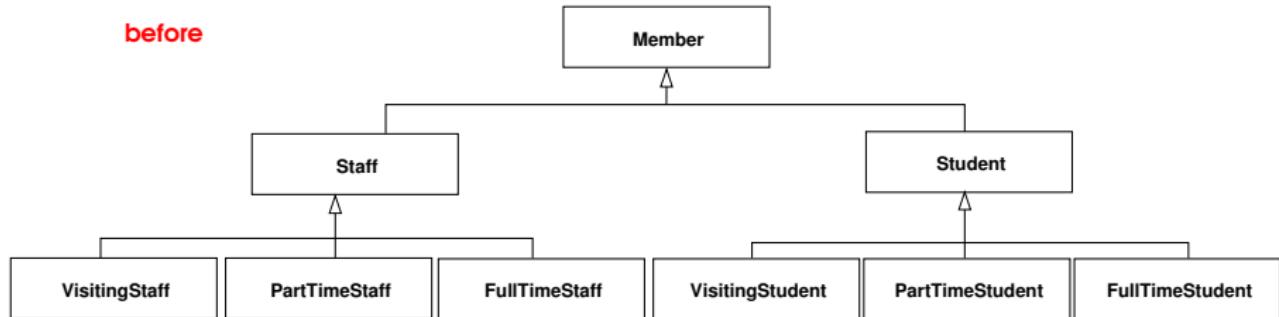


Braude & Bernstein (2011, Figure 24.7)

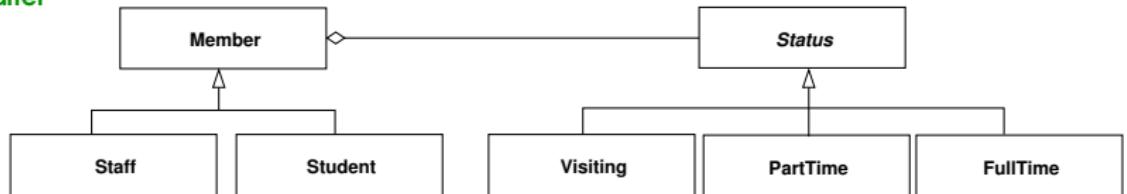
Big Refactorings

2. Tease apart inheritance

- To alleviate the potential for *class explosion*, identify **common properties** and model them as **separate class hierarchies**.



after

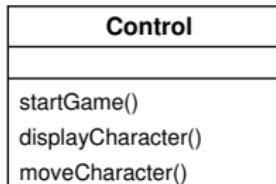


Big Refactorings

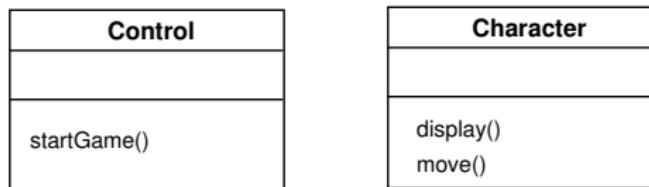
3. Convert procedural design to objects

- ▶ Improve the design by removing *procedural* components into classes.

before



after



A PHP example:

<https://dzone.com/articles/practical-php-refactoring-48>

Big Refactorings

4. Separate domain from presentation

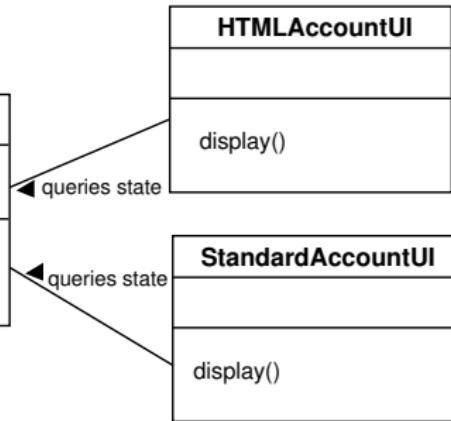
- Make the **responsibility** of each class more **focused** by separating the presentation operations from the domain classes.

before

Account
- name : String
- balance : double
+ deposit(amount:int)
+ displayHTML()
+ displayStandard()

after

Account
- name : String
- balance : double
+ deposit(amount:int)
+ withdraw(amount:int)



Braude & Bernstein (2011, Figure 24.6)

Other Types of Refactoring

Composing Methods

These refactorings apply at **method level only**.

► Extract method

- ▶ identify a block of code and *create a new method* to contain it
- ▶ Alt + Shift + M in Eclipse
- ▶ Why?
 - ▶ Code becomes more readable, especially if the new method has a meaningful name.
 - ▶ Code becomes more modular and re-usable if that block of code is generically useful

Other Types of Refactoring

Composing Methods

These refactorings apply at **method level only**.

- ▶ **Inline method**
 - ▶ *replace* a call to a method by the statements of the method
 - ▶ Alt + Shift + I in Eclipse
 - ▶ Why?
 - ▶ e.g. where a method body has become condensed down to just one line..
 - ▶ Inverse of *Extract method!*

Other Types of Refactoring

Composing Methods

- ▶ **Inline temp**

- ▶ remove a **local temporary** variable and replace it by the associating expression

```
boolean hasDiscount(Order order) {  
    double basePrice = order.basePrice();  
    return basePrice > 1000;  
}
```

Other Types of Refactoring

Composing Methods

- ▶ **Introduce explaining variable**
 - ▶ use a local variable to replace a portion of a complex expression
 - ▶ Alt + Shift + I in Eclipse

```
function Price(Quantity, ItemPrice: Integer): Extended;  
begin  
    Result := (Quantity * ItemPrice) -  
              (MaxInt(0, (Quantity - 500)) * ItemPrice * 0.05);  
end;
```

Other Types of Refactoring

Composing Methods

► Introduce explaining variable

```
function Price(Quantity, ItemPrice: Integer): Extended;
begin
    Result := (Quantity * ItemPrice) -
        (MaxInt(0, (Quantity - 500)) * ItemPrice * 0.05);
end;
```

```
function Price(Quantity, ItemPrice: Integer): Extended;
var
    Discount: Extended;
begin
    Discount := (MaxInt(0, (Quantity - 500)) * ItemPrice * 0.05);
    Result := (Quantity * ItemPrice) - Discount;
end;
```

Other Types of Refactoring

Moving Features between Objects

► **Move method**

- ▶ move a method from one class to another
- ▶ Alt + Shift + V in Eclipse

► **Move field**

- ▶ when a new class is introduced, existing fields may need to be moved to the new class
- ▶ Alt + Shift + V in Eclipse

Other Types of Refactoring

Moving Features between Objects

- ▶ **Extract class**
 - ▶ make a new class from a subset of existing fields and methods
 - ▶ Extract Class in Eclipse

Other Types of Refactoring

Dealing with Generalisation

► Pull Up Field

- When ≥ 2 subclasses in the same class hierarchy have the same field(s), move the common field(s) from the subclasses to the superclass.

- Pull Up in Eclipse

► Pull Up Method

- When two methods in different subclasses within the same class hierarchy have essentially the same behaviours, move the method to the superclass.

- Pull Up in Eclipse

Other Types of Refactoring

Dealing with Generalisation

► **Push Down Field**

- ▶ When a field is used by some subclasses only, but not all subclasses, move the field to those subclasses which use that field.
- ▶ Push Down in [Eclipse](#)

► **Push Down Method**

- ▶ When there is a method in a superclass that is relevant to only some its subclasses, move that method to those subclasses.
- ▶ Push Down in [Eclipse](#)

Other Types of Refactoring

Dealing with Generalisation

► Extract Subclass

- When there is a feature in a class that is applicable to some instances only, create a subclass for that class and move the feature to it.

► Extract Superclass

- make a new class from the common field(s) and method(s) in a set of existing classes, with the extracting classes becoming the direct subclasses of the new class
- Extract Superclass in [Eclipse](#)

Other Types of Refactoring

Dealing with Generalisation

► Extract Interface

- ▶ make a new interface from an existing class, with the class becoming one which implements the new interface
- ▶ Extract Interface in Eclipse

For detail, see the Eclipse help manual on **Refactor Actions** at help.eclipse.org/latest/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Freference%2Fref-menu-refactor.htm

Code smells

- ▶ "Smells are certain structures in the code that indicate violation of fundamental design principles and negatively impact design quality". Suryanarayana et al. 2014,
https://books.google.co.uk/books/about/Refactoring_for_Software_Design_Smells.html
- ▶ Can be useful catalysts for refactoring, especially in an Agile approach, e.g..
 - ▶ *Middle Man*
 - ▶ *Feature Envy*
 - ▶ *Long Parameter List*
 - ▶ *Speculative Generality -> Dead Code...*

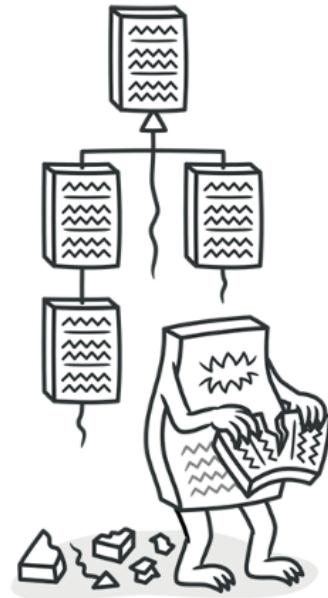
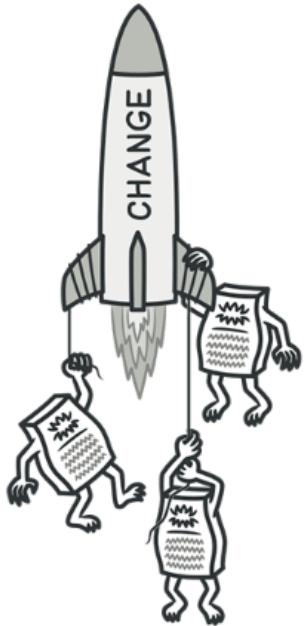
Change Preventers

These smells mean that if you need to change something in one place in your code, you have to make many changes in other places too. Program development becomes much more complicated and expensive as a result.

§ [Divergent Change](#)

§ [Parallel Inheritance
Hierarchies](#)

§ [Shotgun Surgery](#)



Object-Orientation Abusers

All these smells are incomplete or incorrect application of object-oriented programming principles.

§ [Alternative Classes
with Different
Interfaces](#)

§ [Refused Bequest](#)
§ [Switch Statements](#)

§ [Temporary Field](#)

Refused Bequest

Signs and Symptoms

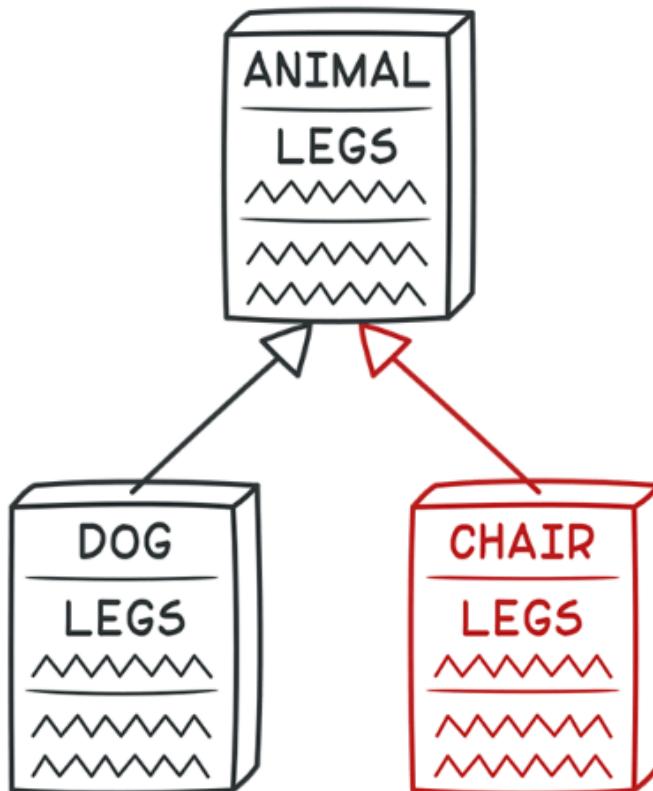
If a subclass uses only some of the methods and properties inherited from its parents, the hierarchy is off-kilter. The unneeded methods may simply go unused or be redefined and give off exceptions.

<https://refactoring.guru/smells/refused-bequest>

Q. What types of coupling and cohesion are related to this?

Reasons for the Problem

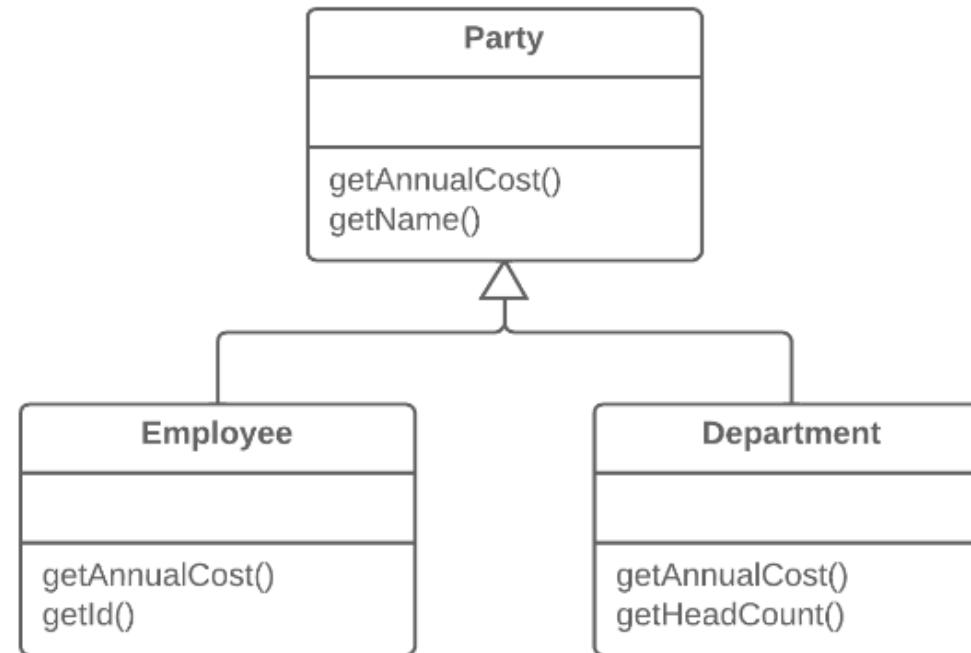
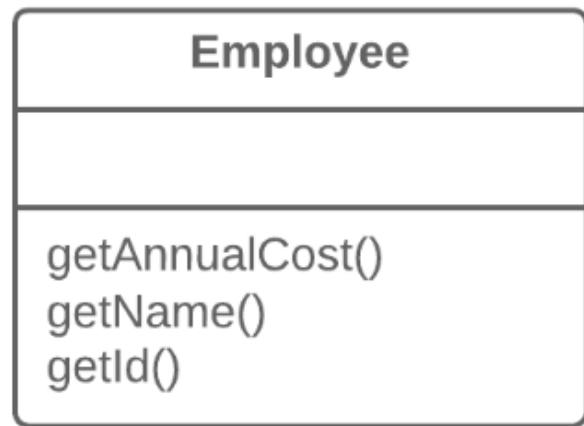
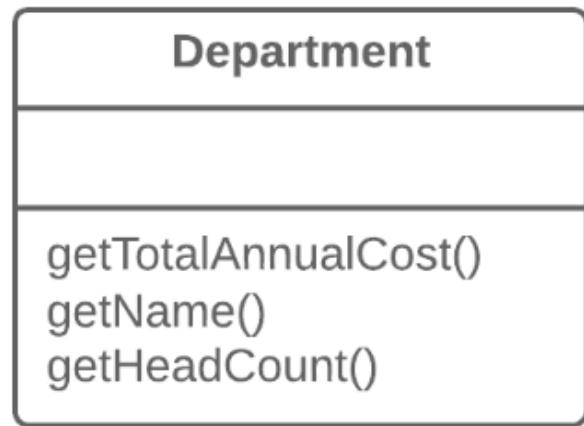
Someone was motivated to create inheritance between classes only by the desire to reuse the code in a superclass. But the superclass and subclass are completely different.



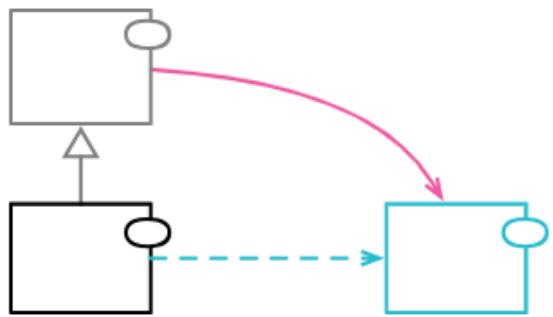
Treatment

- If inheritance makes no sense and the subclass really does have nothing in common with the superclass, eliminate inheritance in favor of Replace Inheritance with Delegation.
- If inheritance is appropriate, get rid of unneeded fields and methods in the subclass. Extract all fields and methods needed by the subclass from the parent class, put them in a new superclass, and set both classes to inherit from it (Extract Superclass).

<https://refactoring.guru/smells/refused-bequest>



Extract Superclass



```
class List {...}  
class Stack extends List {...}
```



```
class Stack {  
    constructor() {  
        this._storage = new List();  
    }  
}  
class List {...}
```

Example from Martin Fowler
<https://refactoring.com/catalog/replaceSuperclassWithDelegate.html>

Instead of inheriting from a superclass, we use composition: the original subclass has an instance of the original superclass, and delegates method calls to it.



The Replace Inheritance with Delegation Refactoring (RIWD)

```
class Client {  
    Stack s = new Stack();  
    ...  
    s.push(...);  
    if (s.size() ...  
}  
  
public class Stack extends Vector {  
    public Stack() {}  
    public Object push(Object item) {  
        addElement(item);  
        return item;  
    }  
}
```

Purpose

to allow reuse without subtyping; to avoid bloated class interfaces; to publicize the interface between a class and its superclass



After the refactoring, the code for `Stack` looks as follows:

```
public class Stack {  
    protected Vector delegatee;  
    public Stack() {  
        delegatee = new Vector();  
    }  
    public Object push(Object item) {  
        delegatee.addElement(item);  
        return item;  
    }  
    public int size() {  
        return delegatee.size();  
    }  
}
```

Code smells

- ▶ "Smells are certain structures in the code that indicate violation of fundamental design principles and negatively impact design quality". Suryanarayana et al. 2014,
- ▶ *Middle Man*
- ▶ *Feature Envy*
- ▶ ***Long Parameter List***
- ▶ *Speculative Generality -> Dead Code...*



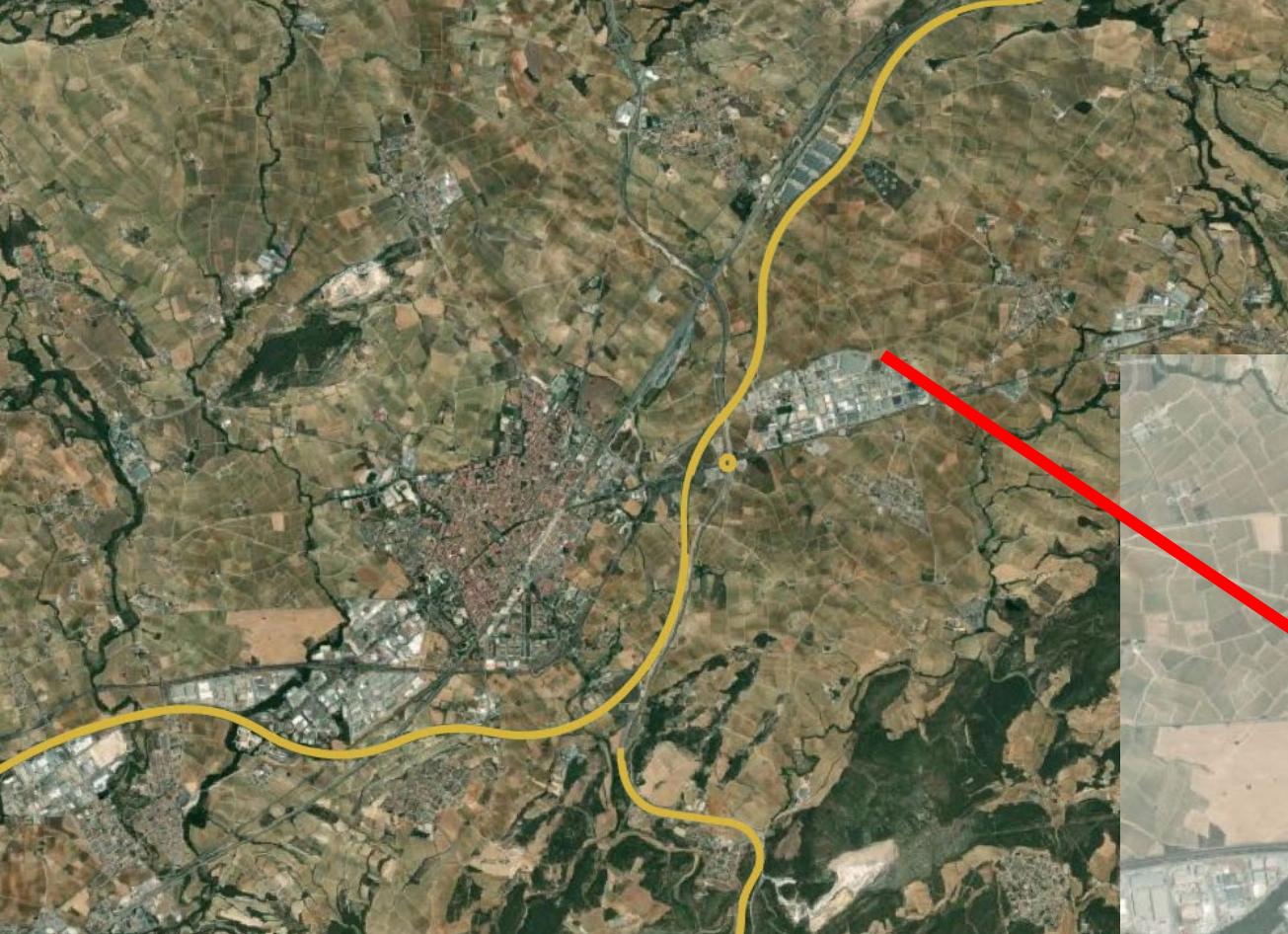
GDAL - Geospatial Data Abstraction Library

GDAL is an open source library for raster and vector geospatial data formats with many useful command line utilities for data translation and processing.

Underpins many applications including NASA Worldwind, Demeter (an OpenGL terrain engine), ESA COPERNICUS data applications, Google Earth, GeoDjango..

https://gdal.org/software_using_gdal.html#software-using-gdal

Written/compiled in C++, with wrappers in many languages including python.



gdal_rasterize



gdal_rasterize

Burns vector geometries into a raster.

Synopsis

```
gdal_rasterize [--help] [--help-general]
[-b <band>]... [-i] [-at]
[-oo <NAME>=<VALUE>]...
{[-burn <value>]... | [-a <attribute_name>] | [-3d]} [-add]
[-l <layername>]... [-where <expression>] [-sql <select_statement>|@<filename>]
[-dialect <dialect>] [-of <format>] [-a_srs <srs_def>] [-to <NAME>=<VALUE>]...
[-co <NAME>=<VALUE>]... [-a_nodata <value>] [-init <value>]...
[-te <xmin> <ymin> <xmax> <ymax>] [-tr <xres> <yres>] [-tap] [-ts <width> <height>]
[-ot {Byte/Int8/UInt16/UInt32/Int32/UInt64/Int64/Float32/Float64/
      CInt16/CInt32/CFloat32/CFloat64}] [-optim {AUTO|VECTOR|RASTER}] [-q]
<src_datasource> <dst_filename>
```

Description

This program burns vector geometries (points, lines, and polygons) into the raster band(s) of a raster image. Vectors are read from OGR supported vector formats.

```
osgeo.gdal.RasterizeOptions(options=None, format=None, outputType=0, creationOptions=None, noData=None, initValues=None, outputBounds=None, outputSRS=None, transformerOptions=None, width=None, height=None, xRes=None, yRes=None, targetAlignedPixels=False, bands=None, inverse=False, allTouched=False, burnValues=None, attribute=None, useZ=False, layers=None, SQLStatement=None, SQLDialect=None, where=None, optim=None, add=None, callback=None, callback_data=None)
```

Create a RasterizeOptions() object that can be passed to gdal.Rasterize()

- Parameters:
- **options** -- can be an array of strings, a string or let empty and filled from other keywords.
 - **format** -- output format ("GTiff", etc...)
 - **outputType** -- output type (gdalconst.GDT_Byte, etc...)
 - **creationOptions** -- list or dict of creation options
 - **outputBounds** -- assigned output bounds: [minx, miny, maxx, maxy]
 - **outputSRS** -- assigned output SRS
 - **transformerOptions** -- list or dict of transformer options
 - **width** -- width of the output raster in pixel
 - **height** -- height of the output raster in pixel
 - **xRes** -- output resolution in target SRS
 - **yRes** -- output resolution in target SRS
 - **targetAlignedPixels** -- whether to force output bounds to be multiple of output resolution
 - **noData** -- nodata value
 - **initValues** --

Value or list of values to pre-initialize the output image bands with.

```
1 if field_name:  
2 # This will rasterize your shape file according to the specified attribute field  
3     rasDs = gdal.Rasterize(rasterized_features, features_to_rasterize, attribute=field_name,  
4                             creationOptions=['COMPRESS=DEFLATE'],  
5                             outputType=gdal.GDT_Int16,  
6                             outputBounds=[x_min, y_min, x_max, y_max],  
7                             outputSRS=inp_srs,  
8                             noData=NoData_value, initValues=NoData_value,  
9                             xRes=cellsize, yRes=-cellsize,  
10                            allTouched=True)  
11  
12 else:  
13 # This will just give 255 where there are vector data since no attribute is defined  
14     rasDs = gdal.Rasterize(rasterized_features, features_to_rasterize, burnValues=255,  
15                             creationOptions=['COMPRESS=DEFLATE'],  
16                             outputType=gdal.GDT_Int16,  
17                             outputBounds=[x_min, y_min, x_max, y_max],  
18                             outputSRS=inp_srs,  
19                             noData=NoData_value, initValues=NoData_value,  
20                             xRes=cellsize, yRes=-cellsize,  
21                            allTouched=True)
```

```
ras_options = gdal.RasterizeOptions(
    creationOptions=['COMPRESS=DEFLATE'],
    outputType=gdal.GDT_Int16,
    outputBounds=[x_min, y_min, x_max, y_max],
    outputSRS=inp_srs,
    noData=NoData_value, initValues=NoData_value,
    xRes=cellsize, yRes=-cellsize,
    allTouched=True)

if field_name:
    # This will rasterize your shape file according to the specified attribute field
    rasDs = gdal.Rasterize(rasterized_features, features_to_rasterize, attribute=field_name, options=ras_options)

else:
    # This will just give 255 where there are vector data since no attribute is defined
    rasDs = gdal.Rasterize(rasterized_features, features_to_rasterize, burnValues=255, options=ras_options)
```

Anti-patterns

- ▶ "Anti-patterns in software engineering, project management, and business processes are 'common responses to solving recurring problems that prove to be ineffective' and are often actually counterproductive. Scott, 1998,
<https://books.google.com/books?id=qJJk2yEeoZoC&q=%22anti-pattern%22+date:1990-2001&pg=PA4>
- ▶ e.g. "A *Big Ball of Mud* is a haphazardly structured, sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle. These systems show unmistakable signs of unregulated growth, and repeated, expedient repair. Information is shared promiscuously among distant elements of the system, often to the point where nearly all the important information becomes global or duplicated." Foote and Yoder, 1999.

<http://www.laputan.org/pub/foote/mud.pdf>

References

- ▶ Braude, E.J. and Bernstein, M.E., *Software Engineering: Modern Approaches*, 2nd ed, Hoboken, NJ: Wiley, 2011.
- ▶ Fowler, M., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999. Also available from Safari Books Online: <http://proquest.safaribooksonline.com/book/software-engineering-and-development/refactoring/0201485672> (06-12-2017).
- ▶ Eclipse, *Eclipse documentation - Current Release*, 2016. (Online). Available at http://help.eclipse.org/oxygen/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Freference%2Fref-menu-refactor.htm&cp=1_4_4_0 (06-12-2017).

For detail, see:

Chapter 24 of Braude & Bernstein (2011).

Learning Outcomes

Learning Outcomes. You should now be able to

- ▶ state the purpose of refactoring
- ▶ explain why refactoring is important in agile development process
- ▶ state various types of refactorings and recognise when to apply them
- ▶ use [Eclipse](#) to perform some refactorings