Specify requirements at the required level of detail, the desired properties and features of the system to be developed.
Three Perspectives on a Solution

• **Data perspective**
  – Data/information to be managed by the software intensive system
  – Static aspects – as entities, relationships between entities, attributes and attribute types

• **Functional perspective**
  – Processes (functions) to be provided by the system
  – Manipulation of the data in each process
  – Input-output relationship among the process

• **Behavioural perspective**
  – States in which the system can be
  – Allowed transitions between the states

---

Goals, Scenarios and Solution-oriented Requirements

![Diagram showing the relationship between Level of agreement, Different views, Consolidated views, Goals, Scenarios, and Solution oriented requirements.]
Goal-Scenario coupling

... initiate and influence the definition of ...

... classify ...

Goals

Scenarios

... illustrate satisfaction...

... lead to the identification of new ...

... lead to revision of ...

Key Relationships

Elicitation, refinement and validation of solution-oriented requirements

Refinement of existing and elicitation of new goals and scenarios
Documenting Solution-Oriented Requirements

Requirements Engineering

Modelling

Partially based on Prof. Steve Easterbrook lecturers on Requirements Engineering, University of Toronto

KAOS – A. van Lamsweerde book:

"Requirements engineering: from system goals to UML models to software specifications"
What is Modelling?
Modelling...

- **Modelling can guide elicitation:**
  - It can help you figure out what questions to ask
  - It can help to surface hidden requirements
    - i.e. does it help you ask the right questions?

- **Modelling can provide a measure of progress:**
  - Completeness of the models -> completeness of the elicitation (?
    - i.e. if we’ve filled in all the pieces of the models, are we done?

- **Modelling can help to uncover problems**
  - Inconsistency in the models can reveal interesting things...
    - e.g. conflicting or infeasible requirements
    - e.g. confusion over terminology, scope, etc
    - e.g. disagreements between stakeholders

- **Modelling can help us check our understanding**
  - Reason over the model to understand its consequences
    - Does it have the properties we expect?
  - Animate the model to help us visualise/validate the requirements

---

**Systems involves a lot of modelling**

- A model is more than just a description
  - It has its own phenomena, and its own relationships among those phenomena.
    - The model is only useful if the model’s phenomena correspond in a systematic way to the phenomena of the domain being modelled

---

Source: Adapted from Jackson, 1995, p120-122
“It’s only a model”

- **There will always be:**
  - phenomena in the model that are not present in the application domain
  - phenomena in the application domain that are not in the model

- **A model is never perfect**
  - “If the map and the terrain disagree, believe the terrain”
  - Perfecting the model is not always a good use of your time...

Sources: Adapted from Jackson, 1995, p124-5

---

Don’t forget what we’re modelling

- **During analysis**
  - we want to know about the application domain and the requirements
  - ...so we develop a course-grained model to show where responsibilities are, and how objects interact
    - Our models show a message being passed, but we don’t worry too much about the contents of each message
    - To keep things clear, use icons to represent external objects and actors, and boxes to represent system objects

- **During design**
  - we want to say how the software should work
  - ... so we develop fine-grained models to show exactly what will happen when the system runs
    - e.g. show the precise details of each method call
KAOS
Goal and Agent (responsibility) model

Elicit Operations
When the refinement should stop?

Elicit Operations

KAOS

Constructs of Goal and Agent models

- **Agent**
  - Active object which plays a specific role towards goal achievement by monitoring or controlling specific object behavior

- **Assignment**
  - A possible assignment of a goal to an agent
  - **Responsibility** – an actual assignment of a goal to an agent

- A goal effectively assigned to
  - A **software agent** is called **requirement**
  - An **environment agent** is called **expectation**
Responsibility assignment

• Refine goals into subgoals
  • Latter require the cooperation of fewer agents

• Stop refining a goal
  – Goal is assigned as the responsibility of a single agent

• Alternative goal responsibility assignments
  – Different design of system-to-be

KAOS
Constructs of Operation model

• Operation
  – An input/output relation over objects
  – Define state transition

• Operationalisation
  – Relationship between goal and operation

• Performs
  – Agent performs operations
KAOS
Operation model

Requirements Modelling

KAOS
Knowledge Acquisition in automated Specification

UML
Unified Modelling Language

Goal model
Object model
Agent model
Operation model
Class diagrams
UML use cases
State and sequence diagrams
Classes

• **A class describes a group of objects with**
  – similar properties (attributes),
  – common behaviour (operations),
  – common relationships to other objects,
  – and common meaning (“semantics”).

• **Examples**
  – *employee*: has a name, employee# and department; an employee is hired, and fired; an employee works in one or more projects

```
:employee
  name
  employee#
  department

  hire()
  fire()
  assignproject()
```

Finding classes

• **Finding classes source data**
  – Look for nouns and noun phrases in stakeholders’ descriptions of the problem
    • include in the model if they explain the nature or structure of information in the application

• **Finding classes from other sources**
  – Reviewing background information
  – Users and other stakeholders
  – Analysis patterns

• **It’s better to include many candidate classes at first**
  – You can always eliminate them later if they turn out not to be useful
  – Explicitly deciding to discard classes is better than just not thinking about them
Selecting classes

• **Discard classes for concepts which:**
  – Are beyond the scope of the analysis;
  – Refer to the system as a whole;
  – Duplicate other classes;
  – Are too vague or too specific
    • e.g. have too many or too few instances

• External entities that produce or consume information essential to the system should be included as classes

Selecting classes

• Coad & Yourdon’s criteria
  – **Retained information**
    • Will the system need to remember information about this class of objects?
  – **Needed Services**
    • Do objects in this class have identifiable operations that change the values of their attributes?
  – **Multiple Attributes**
    • If the class only has one attribute, it may be better represented as an attribute of another class
  – **Common Attributes**
    • Does the class have attributes that are shared with all instances of its objects?
  – **Common Operations**
    • Does the class have operations that are shared with all instances of its objects?
Objects vs. Classes

- The instances of a class are called objects

<table>
<thead>
<tr>
<th>Fred_Bloggs:Employee</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: Fred Bloggs</td>
</tr>
<tr>
<td>Employee #: 234609234</td>
</tr>
<tr>
<td>Department: Marketing</td>
</tr>
</tbody>
</table>

- Two different objects may have identical attribute values (like two people with identical name and address)

- Objects have associations with other objects
  - E.g. Fred_Bloggs:employee is associated with the KillerApp:project object
  - But we will capture these relationships at the class level (why?)

Associations

- Objects do not exist in isolation from one another
  - A relationship represents a connection among things

- Types of relationships
  - Association
  - Aggregation and Composition
  - Generalization

- Class diagrams show classes and their relationships
Multiplicity

- **Optional** (0 or 1) 0..1
- **Exactly one** 1 = 1..1
- **Zero or more** 0..* = *
- **One or more** 1..*
- **A range of values** 1..6
- **A set of ranges** 1..3, 7..10, 15, 19..*

Class association

- **:StaffMember**
  - staffName
  - staff#
  - staffStartDate
  - contact person 1
- **:Client**
  - companyAddress
  - companyEmail
  - companyFax
  - companyName
  - companyTelephone
  - ClientList 0..*

**Multiplicity**
- A client has exactly one staffmember as a contact person
- A staff member has zero or more clients on his/her clientList

**Name of the association**
- liaises with

**Direction**
- The "liaises with" association should be read in this direction

**Role**
- The staffmember’s role in this association is as a contact person
- The clients’ role in this association is as a clientList
Examples

Association classes

- Sometimes the association is itself a class
  - ...because we need to retain information about the association
  - ...and that information doesn’t naturally live in the classes at the ends of the association
  - E.g. a “title” is an object that represents information about the relationship between an owner and her car
Aggregation and Composition

- **Aggregation**
  - This is the “Has-a” or “Whole/part” relationship

- **Composition**
  - Strong form of aggregation that implies ownership:
    - if the whole is removed from the model, so is the part
    - the whole is responsible for the disposition of its parts

Generalisation

- Subclasses **inherit** attributes, associations, & operations from the superclass
- A subclass may override an inherited aspect
- Superclasses may be declared `{abstract}`, meaning they have no instances
  - Implies that the subclasses cover all possibilities
Generalisation

- **Look for generalisations in two ways**
  - Top Down
    - You have a class, and discover it can be subdivided
  - Bottom Up
    - You notice similarities between classes you have identified

- **But don’t generalise just for the sake of it**
  - Be sure that everything about the superclass applies to the subclasses
  - Don’t add subclasses or superclasses that are not relevant to your analysis

Class diagram
Goal model versus Object model

Goal model versus Object model

Goals concern Objects
Goal model versus **Object model**

- **Goal model**
  - Date
  - Excluded date
  - Available date
  - Preferred date
  - Proposed date
  - Access control list

- **Object model**
  - Set of dates
  - Agreement
  - Encrypted agreement
  - Cryptographic keys
  - Scheduler
  - Database
  - Firewall

Requirements Engineering and Modelling

**KAOS**
- Knowledge Acquisition in automated Specification
  - Goal model
  - Agent model

**UML**
- Unified Modelling Language
  - Object model
  - Operation model
  - Class diagrams
  - State and activity diagrams
  - UML use cases
Moving towards specification

- **What functions will the new system provide?**
  - How will people interact with it?
  - Describe functions from a user’s perspective

- **Use Cases**
  - Used to show:
    - the functions to be provided by the system
    - which actors will use which functions
  - Each Use Case is:
    - a pattern of behavior that the new system is required to exhibit
    - a sequence of related actions performed by an actor and the system via a dialogue

- **An actor:**
  - anything that needs to interact with the system/software:
    - a person
    - a role that different people may play
    - another (external) system/software

Another example

[Diagram of a car fixing system with actors like Driver, Gas Attendant, Mechanic, and actions like Drive, Fill Up, Check Oil, Turn On Engine, Fix Car, Fix Car on the Road]
Identifying Actors

• **Ask the following questions:**
  – Who will be a primary user of the system? (primary actor)
  – Who will need support from the system to do her daily tasks?
  – Who will maintain, administrate, keep the system working? (secondary actor)
  – Which hardware devices does the system need?
  – With which other systems does the system need to interact with?
  – Who or what has an interest in the results that the system produces?

• **Look for:**
  – the users who directly use the system
  – also others who need services from the system

Finding Use Cases

• **For each actor, ask the following questions:**
  – Which functions does the actor require from the system?
  – What does the actor need to do?
  – Does the actor need to read, create, destroy, modify, or store some kinds of information in the system?
  – Does the actor have to be notified about events in the system?
  – Does the actor need to notify the system about something?
  – What do those events require in terms of system functionality?
  – Could the actor’s daily work be simplified or made more efficient through new functions provided by the system?
Getting objects to behave

- **All objects have “state”**
  - The object either exists or it doesn't
  - If it exists, then it has a value for each of its attributes
  - Each possible assignment of values to attributes is a “state”
    - (and non-existence is a state, although we normally ignore it)

- **A stack object**
StateCharts

- **States**
  - “interesting” configurations of the values of an object’s attributes
  - States may be “on” or “off” at any given moment
- **Transitions**
  - Are enabled when the state is “on”; disabled otherwise
  - Every transition has an event that acts as a trigger
  - A transition may also have a condition (or guard)
  - A transitions may also cause some action to be taken
- **Events**
  - occurrence of stimuli that can trigger an object to change its state
  - determine when transitions can fire

States

- **A state represents a time period during which**
  - A predicate is true
    - e.g. (budget - expenses) > 0,
  - An action is being performed, or an event is awaited:
    - e.g. checking inventory for order items
    - e.g. waiting for arrival of a missing order item
- **States can have associated activities:**
  - do/activity
    - carries out some activity for as long as the state is “on”
  - entry/action and exit/action
    - carry out the action whenever the state is entered (exited)
  - include/stateDiagramName
    - “calls” another state diagram, allowing state diagrams to be nested
Events

- **Events**
  - Must be relevant to the system (or object) being modelled
  - Must be modellable as an instantaneous occurrence (from the system’s point of view)
    - E.g. completing an assignment, failing an exam, a system crash
  - Are implemented by message passing in an OO Design

- **Four types of events:**
  - *Change events* occur when a condition becomes true
    - E.g. when \([\text{balance} < 0]\)
  - *Call events* occur when an object receives a call for one of its operations to be performed
  - *Signal events* occur when an object receives an explicit (real-time) signal
  - *Elapsed-time events* mark the passage of a designated period of time
    - E.g. after [10 seconds]

What does the model mean?

- **Finite State Machines**
  - There are a finite number of states (all attributes have finite ranges)
    - E.g. imagine a stack with max length = 3

  ![Finite State Machine Diagram]

  - The model specifies a set of traces
    - E.g. new();Push();Push();Pop();Top();Pop();Push();...
    - E.g. new();Pop();Push();Pop();Pop();...
    - There may be an infinite number of traces (and traces may be of infinite length)

  - The model excludes some behaviours
    - E.g. no trace may have more Pops than Pushes
    - E.g. no trace may have more than 3 Pushes without a Pop in between
Object states

```
:person
  age
  havebirthday()

child
  havebirthday() [age < 18]

adult
  havebirthday() [age < 65]
  havebirthday() [age = 65]

senior
  havebirthday() [age = 65]

:person
  dateOfBirth
  dateOfDeath
  recordBirth()
  setDateOfBirth()
  recordDeath()
  setDateofDeath()

blank
  recordBirth()
  setDateOfBirth()

child
  when [thisyear - birthyear > 18]

adult
  when [thisyear - birthyear > 65]

senior
  recordDeath()
  setDateOfDeath()

deceased
```

Requirements Modelling

**KAOS**
Knowledge Acquisition in automated Specification

- Goal model
- Agent model
- Operation model

**UML**
Unified Modelling Language

- Class diagrams
- UML use cases
- State and sequence diagrams
Sequences of Events

• **Objects “own” information and behaviour**
  – they have attributes and operations relevant to their *responsibilities*
  – They don’t “know” about other objects’ information, but can ask for it.
  – Objects have to collaborate
    • …by sending messages to one another to invoke each others’ operations
  – Objects can only send messages one to another if they “know” each other
    • if there is an association between them

• **Sequence Diagrams**
  – Sequence diagrams show how objects interacts with one another
  – Sequence diagrams...
    • …should remain easy to read and understand
    • …do not include complex control logic

Example

- Initiator: Person
  - Call()
  - What’s up?()
  - Give mtg details()

- Staff Person
  - Respond()
  - [for all participants] *Inform()

- Scheduler Person
  - Acknowledge()

- Participant Person
  - [decision=OK] ScheduleOK’ed()
  - [for all participants] *Inform()
Another example

Campaign Manager

getName()
listCampaigns()
listAdverts()
addNewAdvert()

Object lifeline
Activation
Object creation

Requirements Modelling

**KAOS**
Knowledge Acquisition in automated Specification

- Goal model
- Agent model

**UML**
Unified Modelling Language

- Object model
- Operation model
- Class diagrams
- State and activity diagrams
- UML use cases
Documenting Solution-Oriented Requirements

Entity: entrance door
Entity: glass break detector

Entity-relationship diagram

Data model

Requirement (natural language)

A glass break detector is attached to the entrance door. If the entrance door has been damaged, the system shall enter the alarm state and inform the security company.

Functional model

Transition diagram

Event: entrance door damaged
State: alarm state
Event: inform security company

Data flow diagram

Function: inform security company