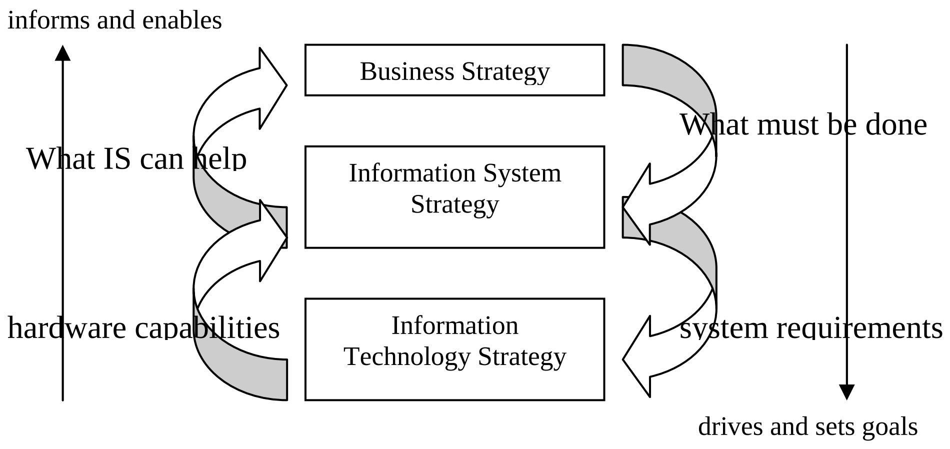
References

1. Object-Oriented Software Engineering---Practical software development using UML and Java, T.C. Lethbridge and R. Laganiere, McGraw-Hill, 2005.
2. Object-Oriented Systems Analysis and Design using UML, Bennett, McRobb, Farmer, McGraw-Hill, second edition, 2002.
3. Object-Oriented Software Engineering---Conquering Complex and Changing Systems, Bernd Bruegge and Allen H. Dutoit, Prentice-Hall International, Inc., 2000.

Information Systems



The relationship between business, IS, and IT strategies

Poor systems

- Poor interface design
- Inappropriate data entry
- Incomprehensible error messages
- Unhelpful 'help'
- Poor response time
- Unreliability in operation

An end-user's perspective

- What system?  I haven't seen a new system

- It might work, but it's dreadful to use!
- It's very pretty, but does it do anything useful?

A client's perspective

- If I'd known the real price, I'd never have agreed
- It's no use delivering it now—we needed it last April!
- OK, so it works—but the installation was such a mess my staff will never trust it
- I didn't want it in the first place
- Everything's changed now—we need a completely different system

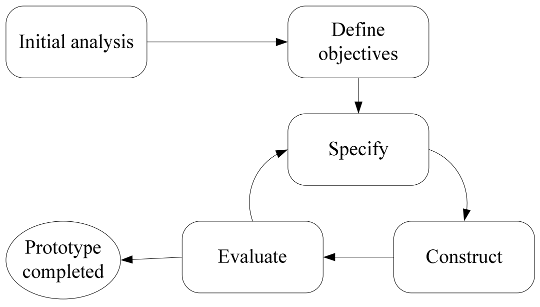
A developer's perspective

- We built what they said they wanted
- There wasn't enough time to do it any better
- Don't blame me—I've never done object-oriented analysis before!
- How can I fix it?—I don't know how it's supposed to work
- We said it was impossible, but no-one listened
- The system's fine—the users are the problem

Why things go wrong

| Type of failure | Reason for failure |
|---|---|
| Quality problems | The wrong problem is addressed |
| | Wider influences are neglected |
| | Analysis is carried out incorrectly |
| | Project undertaken for wrong reason |
| Productivity problems | Users change their minds (requirements drift) |
| | External events change the environment |
| | Implementation is not feasible |
| | Poor project control |

## Prototyping



Pros

- Early demonstrations of system functionality help identify any misunderstandings between developer and client
- Client requirements that have been missed are identified
- Difficulties in the interface can be identified
- The feasibility and usefulness of the system can be tested, even though, by its very nature, the prototype is incomplete

Cons

- The client may perceive the prototype as part of the final system, may not understand the effort that will be required to produce a working production system and may expect delivery soon
- The prototype may divert attention from functional to solely interface issues
- Prototyping requires significant user involvement
- Managing the prototyping life cycle requires careful decision making

## Incremental development

- The production of high-value to low-cost increments
- The delivery of usable increments of 1% to 5% of total project budget
- A limit to the duration of each cycle (e.g. one month)
- A measure of productivity in terms of delivered functionality or quality improvements
- An open-ended architecture that is a basis for further evolutionary development

Example of incremental development: Boehm's Spiral Model

The Unified Development Process

- Inception—scope and purposes of the project
- Elaboration—requirements capture and determining the structure of the system
- Construction—build the software system
- Transition—product installation and rollout

| | Requirements | Analysis | Design | Implement | test |
|---|---|---|---|---|---|
| Inception | XXX | XX | X | X | X |
| | XXXX | XXX | X | X | X |
| Elaboration | XXXX | XXXX | XX | X | X |
| | XXX | XXX | XXXX | XX | X |
| Construction | XX | XXX | XXXX | XXXX | XXX |
| | X | XX | XXX | XXXX | XXX |
| | X | X | XXX | XXXX | XXXX |
| | | X | XX | XXX | XXXX |
| Transition | | X | X | XX | XX |
| | | | X | X | XX |

## Managing IS Development

- User involvement
- Methodological approaches
- CASE (Computer-Aided Software Engineering)

Object: ID, state, behavior

## UML diagrams

Class diagrams: describe the system in terms of objects, classes, attributes, operations, and their associations.

Sequence diagrams: formalize the behavior of the system and visualize the communication among objects.

Statechart diagrams: describe the behavior of an individual object as a number of states and transitions between these states (A state represents a particular set of values for an object)

Activity diagrams: describe a system in terms of activities, which are states that represent the execution of a set of operations. The completion of these operations triggers a transition to another activity.

Relationships: include, extend, generalization, specialization

Associations: role, multiplicity, aggregation

## Requirements elicitation concepts

- Functional requirements
- Nonfunctional requirements
- Levels of descriptions
- Correctness, completeness, consistency, clarity, and realism
- Verifiability and traceability
- Greenfield engineering, reengineering, and interface engineering

Levels of description

1. work division: users and the system
2. application-specific system functions: system functions relate to application domain
3. work-specific system functions: supporting functions that are not directly related to application domain, e.g., management, grouping, undo, …, which will be extended during system design
4. dialog: interactions between users and UI

correctness: complete, consistence, unambiguous, realistic

verifiability: good UI, error free, respond in 1 sec in most cases

Greenfield: from scratch, triggered by a user's need or a new market

Reengineering: redesign and reimplementation of an existing system triggered by technology enablers or new information flow

Interface engineering: …

Requirement elicitation activities

- identify actors
- identify scenarios
- identify use cases
- refining use cases
- identify relationships among use cases
- identify participating objects
- identify nonfunctional requirements

scenarios: a narrative description of what people do and experience as they try to make use of computer systems and applications, consist of as-is, visionary, evaluation, training

gathering requirements

- observation
- interview (and recorded interview)
- brainstorming
- prototyping

factors affecting requirements document

- size of the system
- need to interface to other systems
- target audience
- contractual arrangements for development
- stage in requirements gathering
- level of experience with the domain and the technology
- cost incurred if the requirements are faulty

Reviewing requirements

- have benefits that outweigh the costs of development
- be important for the solution of the current problem
- be expressed using a clear and consistent notation
- be unambiguous
- be logically consistent
- lead to a system of sufficient quality
- be realistic with available resources
- be verifiable
- be uniquely identifiable
- not over-constrain the design of the system
- doc should be sufficiently complete, well-organized, clear reasoning, and agreed to by all the stakeholders

<u>relationship among actors and use cases</u>

- communication relationships between actors and use cases

- extended relationships between use cases
- include relationships between use cases (facto out redundancies)
- extend (condition extension) VS include (condition initiator) relationships

use extends for exceptional, optional, or seldom occurring behavior

use include for behavior that is shared across two or more use cases

<u>participating objects</u>

- correspond to the main concepts in the application domain
- initial analysis objects
    - terms that developers/users need to clarify in order to understand the use cases
    - recurring nouns in the use cases
    - real world entities that the system needs to keep track of
    - use cases
    - data sources or sinks
    - interface artifacts
    - always use application domain terms

    which use cases create this objects?

    which actor can access this information?

    which use cases modify and destroy this object?

    which actor can initiate these use cases?

    Is this object needed?

Nonfunctional requirements

- UI and human factors
- Documentation
- Hardware
- Performance characteristics
- Error handling and extreme conditions
- Quality issues

- System modifications
- Physical environment
- Security issues
- Resource issues

Managing requirement elicitation

- Elicit requirements from users (domain/knowledge)
- Negotiating a spec with clients (joint application design)
- Validating requirements (usability testing)
- Documenting requirements elicitation

Managing changing requirement

- Business process changes
- Technology changes
- Better understanding of the problem
- (beware of requirements creep)

Knowledge Analysis of Tasks (KAT)

- Identify objects and actions
- Identify procedures
- Identify goals and subgoals
- Identify typicality and importance
- Construct a model of a task

Analysis Model

consists of

| | | |
|---|---|---|
| (use case) | (class) | (statechart, sequence,…) |
| functional model | object model | dynamic model |

- Entity, boundary, control objects
- Association multiplicity
- Qualified associations (technique for reducing multiplicity by using keys)
- Generalization

<u>Identify entity objects</u>

| | |
|---|---|
| ● Object | proper noun |
| ● Class | common noun |
| ● Operation | verb (do) |
| ● Inheritance | verb (has, includes) |
| ● Constraints | verb (modal, must be) |
| ● Attributes | adjective |

<u>Modeling integration</u> (diagram)

- Collaboration (communication) diagram
- Sequence
- Timing
- Interaction overview

<u>Behavior</u>: class, activity, statechart

<u>Structure</u>: class, object, component, composition structure, package, deployment

## System Design

analysis results: (from author's point of view) "WHAT"

- A set of nonfunctional requirements and constraints
- A use case model
- An object model
- A sequence diagram

Not "HOW"

- No internal structure of the system
- How the system should be realized

Aspects of software design

- Architecture
- Detailed: class, UI, algorithms
- Database
- Network/protocol

Design principles

1. divide and conquer
2. increase cohesion where possible
3. reduce coupling where possible
4. keep the level of abstraction as high as possible
5. increase reusability where possible
6. reuse existing designs and code where possible
7. design for flexibility
8. anticipate obsolescence
9. design for portability
10. design for testability

<u>Products</u>:

- A list of design goals (qualities of system that developers should optimize)
- Software architecture (system decomposition into subsystem responsibilities, dependencies, mapping to hardware, control flow, access control, data storage)

<u>Details</u>:

1. hardware mapping
2. data management
3. access control
4. control flow
5. boundary condition

<u>software architecture</u>

- repository architecture
- model/view/controller
- client/server
- pipe and filter

<u>design approaches</u>: model driven, aspect-oriented

software architecture contents

- logical breakdown into subsystems
- dynamics of interaction among component at run-time

- data sharing
- components exist at run-time

<u>system design activities</u>

- identify design goals from nonfunctional requirements
- design initial subsystem decomposition
- map subsystem to processors and components
- select a control flow mechanism
- identify boundary conditions

<u>design issues</u>

- design goals: reliability, fault tolerance, security, modifiability
- performance criteria: response time, throughput, memory
- dependability criteria: robustness, reliability, availability, fault tolerance, security, safety
- cost criteria: development, deployment, upgrade, maintenance, admin
- maintenance: extensibility, modifiability, adaptability, portability, readability, traceability (to requirements)
- end user criteria: utility, usability

design goal trade-offs: space VS speed, delivery time VS (functionality, quality, staffing)

<u>example</u>

files VS database

when to choose a file?

- voluminous data (images)
- temp data (core)
- low information density (archival, history logs)

when to choose a database?

- Concurrent access
- Access at a fine level of details
- Multiple platforms
- Multiple applications over the same data

when to choose relational DB?

- Complex queries over attributes
- Large datasets

when to choose OODB?

- Extensive use of associations to retrieve data
- Medium-sized data set
- Irregular associations among objects

<u>Example cases</u>

1. multi-layer architectural patterns
2. client-server and other distributed architectural patterns
3. broker architectural patterns
4. transaction processing architectural patterns
5. pipe and filter architectural patterns
6. model-driven controller architectural patterns
7. service-oriented architectural patterns
8. message-oriented architectural patterns

<u>characteristics of good design</u>

-hierarchical organization

-modular

-abstraction (data & process)

-independent functional characteristic

-simple interface

-repeatable

<u>issues concerning modular design:</u>

| | |
|---|---|
| objectives: | architectural design |
| | module design |
| | debugging |
| | testing |
| | maintenance |

team programming

reuse

| | | |
|---|---|---|
| modularity: | -how big should a module be? | (size) |
| | -how complex is this module? | (complexity) |
| | -how can we minimize interactions between modules? (low coupling) | |

<u>Modularization criteria</u>

1. cohesion
   - coincidental: brought together into a single component
   - logical: similar functionality (output to screen/printer/file)
   - temporal: activated at a single time (initialization)
   - procedural: components make up a single control sequence
   - communicational: operate on same input/output data
   - sequential: output of one component becomes input of the next
   - functional: each part is necessary for the execution of a single function
2. coupling
   - content: one module modifies local data values or instructions in another module (assembly programs)
   - common: tightly coupled – shared variables, interchange control information (global)
   - control: tightly coupled – (flag)
   - stamp: loosely coupled – within parameter list, sharing in the form of package (struct)
   - data: loosely coupled – (single data)
3. understandability
   - cohesion
   - naming
   - documentation
   - complexity
4. adaptability: (easy to make design change) high level of visibility, clear relationship between different levels of design

## Design Patterns

1. abstraction-occurrence pattern

   Context: Often in a domain model you find a set of related objects (occurrences). The members of such a set share common information, but also differ from each other in important ways.

2. general hierarchy pattern

   Context: Objects in a hierarchy can have one or more objects above them (superiors), and one or more objects below them (subordinates). Some objects cannot have any subordinates (not inheritance hierarchy)

3. player-role pattern

   Context: A role is a particular set of properties associated with an object in a particular context. An object may play different roles in different contexts.

4. singleton pattern

   Context: It is very common to find classes for which only one instance should exist (singleton)

5. observer pattern

   Context: When an association is created between two classes, the code for the classes becomes inseparable. If you want to reuse one class, then you also have to reuse the other.

6. delegation pattern

   Context: You are designing a method in a class. You realize that another class has a method which provides the required service. Inheritance is not appropriate, E.g. because the isa rule does not apply

7. adapter pattern

   Context: You are building an inheritance hierarchy and want to incorporate it into an existing class. The reused class is also often already part of its own inheritance hierarchy.

8. façade pattern

   Context: Often, an application contains several complex packages. A programmer working with such packages has to manipulate many different classes

9. immutable pattern

Context: An immutable object is an object that has a state that never changes after creation.

10. read-only interface pattern

Context: You sometimes want certain privileged classes to be able to modify attributes of objects that are otherwise immutable.

11. proxy pattern

Context: Class diagrams show how aspects of the architecture of a system will be implemented. Often, it is time-consuming and complicated to create instances of a class (heavyweight classes). There is a time delay and a complex mechanism involved in creating the object in memory

12. factory pattern

Context: A reusable framework needs to create objects as part of its work. However, the class of the created objects will depend on the application.

Object design

1. service spec
2. component selection
3. restructuring
4. optimization

during analysis→attrib, operation (together form object type)→visibility (used by other classes: pub, pro, pri)

contracts: invariants (predicate that is always true for all instances of a class), preconditions, postconditions

object design (Object Constraint Language)

- spec -attrib and operations
  -type signatures and visibility
  -constraints
  -exceptions

- component selection
    - -identify and adjust class libraries
    - -identify and adjust application frameworks
- restructuring
    - -realizing associations
    - -Increasing reuse
    - -removing implementation dependencies
- optimization
    - -revisiting access paths
    - -collapsing objects (tuning object into attrib)
    - -caching the result of expensive computations
    - -delaying expensive computations

<u>Managing object design</u>

- increase communication complexity
- consistency with prior decisions and document

<u>Documenting object design</u>

- restrictiveness
- generality
- clarity
- self-contained object design document (ODD) generated from model
- ODD as extension of the RAD
- ODD embedded into source code

## Testing

failure, defect, error, WB/BB (all possible paths, all possible edges, all nodes), equivalent partitioning, combinations of equivalence classes

<u>program defects</u>:

1. incorrect logical operations
2. loops
3. recursion

4. preconditions
5. null conditions
6. singleton/non-singleton conditions
7. off-by-one errors
8. operator precedence errors
9. use in appropriate standard algorithms

<u>defect numerical algorithm</u>

- not using enough bits or digits to store max values
- using insufficient places after decimal point or too few significant figures
- ordering operations poorly so that errors build up
- assuming a floating point value will be exactly equal to some other values

<u>timing and coordination</u>

- deadlock and livelock
- critical races

<u>stress unusual conditions</u>

- insufficient throughput or response time on minimal configurations
- defects in handling peak loads or mission resources
- inappropriate management of resources
- defects in the process of recovering from a scratch

<u>document defects</u>

- writing formal test cases and test plans
- strategies: TD, BU, Big-bang, Sandwich, test-fix-test, cycle/regression, alpha, beta, acceptance

<u>inspection</u>: author, moderator, secretary, paraphrasers

<u>peer review</u>: quality assurance, root cause analysis, continuous improvement, post-mortem analysis, process standards

## Project management

- process models
    1. waterfall
    2. phased-release
    3. spiral
    4. evolutionary
    5. concurrent engineering
    6. rational unified process
    7. agile
    8. aspect-oriented software development
    9. open source
- reengineering, cost estimation, team structures

members: architect, project manager, CM and build specialist, UI specialist, technology specialist, HW and third-party, SW specialist, user documentation specialist (technical writer), tester.

Techniques: GANTT, CPM, PERT, Earned value charts.