COMP354

Software Engineering

Lecture 10–11

Design Overview

Outline

Design process

- Design phases
- Deliverables
- Activities

DESIGN FOR CHANGE

- The nature of change
- Information hiding, Modularization

How to review designs

Architectures

Design patterns

Design produces a solution meeting the functional and non-functional constraints of the requirements.

This is called *"fitness for purpose"*

The solution describes **how** to do the task.

A "good" design should provide
- fitness for purpose (correct, reliable, robust)
- maintainable (design for change)
- "positive" qualities important to user

Design Phases

Architectural Design looks at structural issues
- organization into subsystems and modules
- assignment of functionality to components
- distribution of control
- protocols for communication, synchronization, and data access
- physical allocation of components to processors

Deliverables: *Architectural Design* (AD);
Interface Specifications (IS)

Detailed Design provides internal details of each module in the design, including
- each routine of interface,
- parameters for each routine,
- format of any input/output,
- the data structures and algorithms used,

Deliverables: *Detailed Design* (DD)

Terminology

Subsystem is subset of the modules making up a system.

Can think of a subsystem as a high-level module,
so a subsystem is a provider of services

Module is a provider of computational resources or services

Can think of a class as a module.

Unit is an individual routine, procedure, function

USES relation: M1 *uses* M2 if, in order for M1 to provide its services, M1 uses the services of M2

IS_COMPONENT_OF relation: M1 *is_component_of* M2 if M1 is physically part of M2

also called *is_part_of* relation

inverse of *aggregation* relation

IS_A relation: M1 *is_a* M2 if M1 can be substituted for M2 wherever M2 is used

also called *inherits_from* or *specialization* relation

inverse of *generalization*

Design Deliverables

Architectural Design (AD) provides
- a **subsystem and module diagram**,
- a brief description of the **role** of each module,
- collaboration between modules, and
- **traceability** information between requirements and the module functionality.

Interface Specifications (IS) describes each service provided by each module.
To specify a function, give:
- name;
- argument types;
- a **requires clause** — a condition that must be true on entry to the function (*pre-condition*);
- an **ensures clause** — a condition that will be true on exit from the function (*post-condition*);
- further comments as necessary.

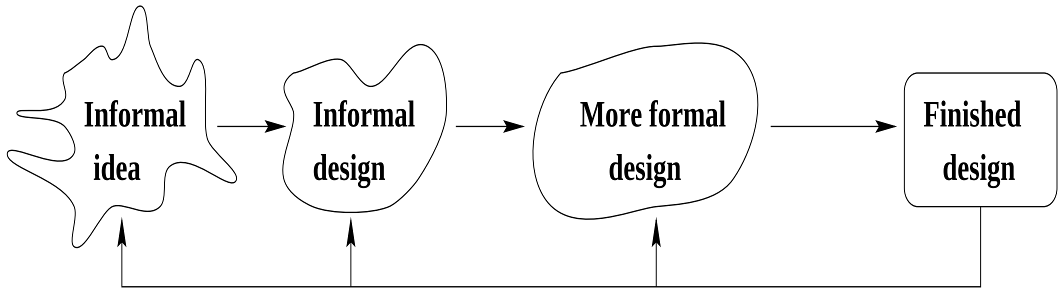
The requires and ensures clause constitute a *contract* between the user and implementor of the function.

Detailed Class Design (DD) has the same structure as the IS of the class, but adds:
- data descriptions (e.g. binary search tree);
- data declarations (types and names);
- a description of how each function will work (pseudocode, algorithm, narrative, . . .).

Design Activities

Basics: *iterate* the following steps

1. propose a tentative design
2. describe as rigorously as necessary
3. trace scenarios to find pitfalls in proposal
4. propose solutions to pitfalls, discuss pros and cons



Brainstorming is team discussion to toss around ideas
— free unconstrained flow of ideas
— no criticism of ideas allowed
— collect as many ideas as possible

Follow-up meeting examines, criticises, and selects ideas

Issue-driven design carefully documents **rationale** for design by examining *issues*

For each issue/problem with proposed design:

- describe issue fully
- list alternative solutions
- argue pros and cons of each solution
- propose new solution combining best ideas from all solutions offered
- iterate
- decide on solution to the issue/problem

Sometimes issue "remains on the table"
while further investigation is done, eg prototypes

Design Activities Overview

decompose system into subsystems and modules
“execute” scenarios of software use cases, to clarify

- which module does what
- which modules are needed
- the use they make of each other
- their raison d’etre
- their responsibilities

iterate until AD document is stable

develop a tentative design of module interfaces
“execute” scenarios of software use cases, to clarify

- what is the sequence of calls to the interface
- what information is passed (in either direction)
- are interfaces complete?
  ie each task can be performed
- are interfaces elegant?
  ie can each task be performed easily

iterate until IS has a complete elegant interface for each module
(may have to change AD, redistribute responsibilities)

choose the class internal design to meet design trade-offs using knowledge of data structure and algorithm trade-offs
(may impact IS, require more info passed in interfaces)

May do all activities at once
— work at different levels of abstraction

Design for Change

A **maintainable** system should be

understandable A design must be understood before it can be changed.

Keep things as simple as possible.

Document well. Document top-down. Document rationale.

modular Want a loosely coupled system of cohesive modules.

Ideal: A (anticipated) change to a module should not impact other modules at all.

Cohesion is a property of modules.
A *cohesive* module provides a small number of closely related services.

Coupling is a property of systems.
Modules are *loosely coupled* if the communication between them is simple.

traceable Design changes may come from changes in requirements, or from defect reports of code.

It must be easy to find all parts of the design that correspond to a requirement, or to a piece of code.

The Nature of Change

What can change? Everything and Anything!

change of algorithms — best understood type of change
eg, sort algorithm

change of data representation — about 17
eg, linked list to hash table
Use abstract data types!

change of underlying abstract machine
eg machine defined by system interface to OS, DBMS, Windows
Use layers!

change of peripheral devices — especially for embedded systems
eg, device drivers for terminals, disks, LAN
eg, image recognition input devices
eg, sensor input to process control system

change of social environment
eg, changes in tax legislation, accounting obligations
eg, changes in interest rates
eg, changes in "business rules"

Information Hiding

Information hiding is the main strategy for design for change.

Encapsulate a module by only allowing access to module services via the *interface*.

The *implementation* details are **hidden**.
No other module depends on these details,
so they can change without impacting other modules.

What should be visible in interface?
— as little as possible

What should be hidden?
— as much as possible, especially things likely to change

Module secrets
— data representation, eg symbol table module
— details of abstract machine
    eg, interface to X windows, or to file system
    eg, details of input formats, syntaxes
— etc

Examples of Modules

Example: a utility module for geometry.
Secret: Representations of geometric objects and algorithms for manipulating them.
Interface: Abstract data types such as *Point*, *Line*, *Circle*, etc. Functions such as:
Line makeline (*Point* p_1, p_2)
Point makepoint (*Line* l_1, l_2) *Circle*
makecircle (*Point* c, `float` r)
Implementation: Representations for lines, circles, etc. (In C, these may be exposed in `.h` files. This is unfortunate but unavoidable.) Implementation of each function.

Example: a stack module.
Secret: How stack components are stored (array/list).
Interface: Functions *Create*, *Push*, *Pop*, *Empty*, *Full*.
Implementation: For the array representation, *Full* returns *true* if there are no more array elements available. The list representation returns *false* always (assuming memory is not exhausted — but that is probably a more serious problem).

Example: a screen manager.
Secret: relationship between stored and displayed data.
Invariant: The objects visible on the screen correspond to the stored data.
Interface:

Display:	add object to store and display it.
Delete:	erase object and remove it from store.
Hide:	erase object (but keep in store).
Reveal:	display a hidden object.

Implementation: An indexed container for objects (or pointers to objects) and functions to draw and erase objects.

A Recipe for Module Design

1. Decide on a secret.

2. Review implementation strategies to ensure feasibility.

3. Design the interface.

4. Review the interface.
   Is it too simple or too complex? Is it cohesive?

5. Plan the implementation.
   E.g. choose representations.

6. Review the module.
   - Is it self-contained?
   - Does it use many other modules?
   - Can it accommodate likely changes?
   - Is it too large (consider splitting) or too small (consider merging)?

How to Review Designs

Aim: to discover errors, not fix them

Design Review

- panel members study the design document(s) (or sections)
- mark items on checklist that seem incorrect or need clarification
- panel meets with designers and discuss marked items

Design Walkthrough

- designer explains logic of the design step by step to a panel of peers
- panel asks questions, point out errors, seek clarification

more informal than review

much benefit for designer in the process of articulating and explaining design

Automated cross checking
eg compiler type-checking procedure calls

Metrics

- number of modules
- fan-in, fan-out
- number of variables, routines, and parameters in interface

General Design Checklist

Is each requirement taken into account?

Are all assumptions explicitly stated?
— are the assumptions acceptable?

Are there any limitations and constraints in the design beyond those in the requirements?

Is the design modular?
— and does it conform to local standards (PDL etc)?

Does each rationale for each module provide a clear strong basis for high cohesion?

Are interfaces of each module completely specified?

Have exceptional conditions been handled?

Are the sizes of data structures estimated?
—are there provisions to guard against overflow?

Are there analyses to demonstrate that performance requirements can be met?

AD Checklist

Is each requirement taken into account?

Is the design modular?
— and does it conform to local standards (PDL etc)?

Does each rationale for each module provide a clear strong basis for high cohesion?

IS Checklist

Are interfaces of each module completely specified?

Have exceptional conditions been handled?

Are all pre- and postconditions explicitly stated?
— do they only refer to parameters of the routine?
(and not to global variables or events)

Does IS conform to AD?

Does IS conform to local standards (PDL etc)?

Is the format of data consistent with externally specified input/output formats?

DD Checklist

Does each module in the design exist in the DD?

Are interfaces in DD consistent with IS?

Is each statement in the DD easily codable?

Are the loop termination conditions clearly stated?

Are conditions in loops and if-statements OK?

Is the module logic too complex?

Is the nesting proper?

Do references to routines and variables in the same module indicate high cohesion?

Do references to routines and variables in the each other module indicate low coupling?

Are the sizes of data structures estimated?
— are there provisions to guard against overflow?

Are there analyses to demonstrate that performance requirements can be met?

COMP354

Software Engineering

Lecture 12

Design Documents

OO Arcitectural Design Document

This document describes the architecture of the system.

The models of this document extend those of the OO Analysis Document by adding “solution-specific” classes and providing more detail (= the design decisions).

Rationale summarises the main issues and their selected solutions.

Architecture is a high-level decomposition of the system into subsystems, with the *system topology* describing the client-supplier and peer-to-peer relationships between subsystems.

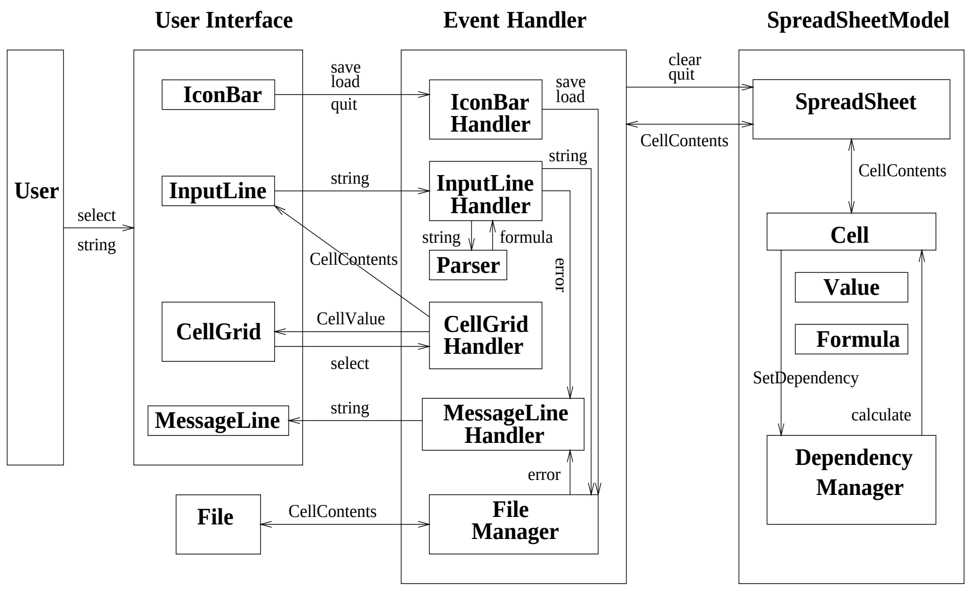
Object Model records the *entities* and their *relationships*. The entities are organised into *classes*, and allocated to subsystems.
There is an overview diagram, followed by several diagrams detailing *attributes* and *interfaces* of major classes.

Dynamic Model records when, and in what order, *events* occur as the user interacts with the system. Events may be calls to *interface* of a class.

Functional Model records the dependencies between each *function* (or process) of the system and the *dataflows* which are the inputs and outputs of the functions.

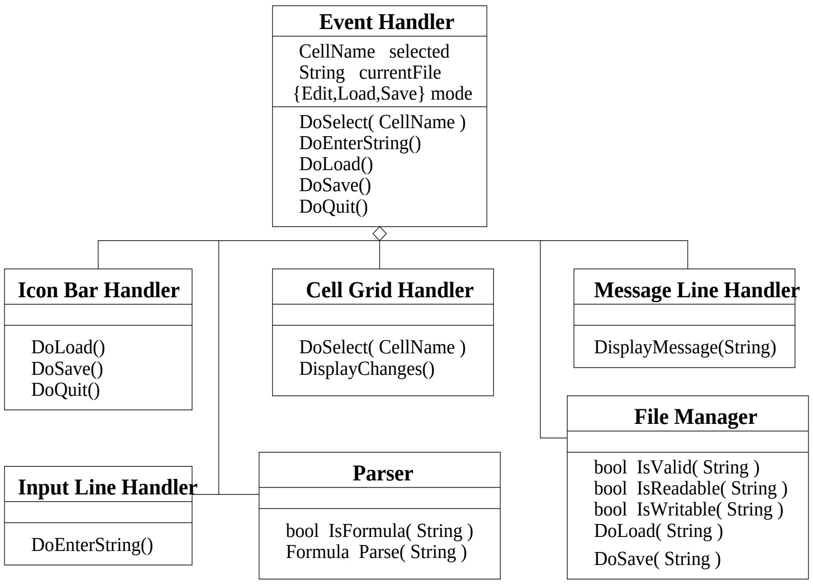
Data Dictionary records definitions of terms used in the above models.

Architecture

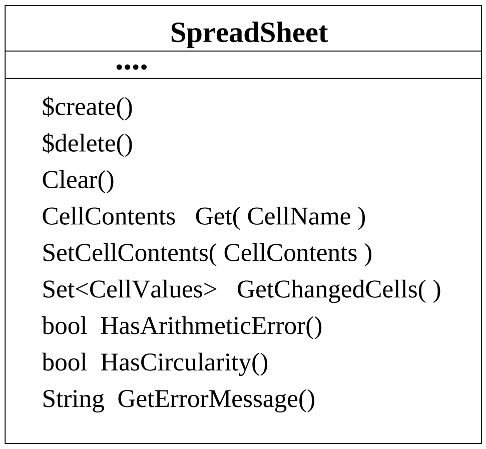


- Subsystem: a major component of a system; provides a service; a package of interrelated classes, associations, ...; has a small well-defined interface
- Client/Supplier: C *uses* S
- Peer-to-Peer: P2 *uses* P2 $\wedge$ P2 *uses* P1
  direct or indirect use; cycles $\Rightarrow$ hard to understand
- Layer: supplier to layer above/client to layer below
- Partition: loosely-coupled peer-to-peer subsystems
- System Topology: the structure of the dependencies (including uses relation and information flow and control flow) between subsystems

Object Model



Note description of *attributes* and *interface*

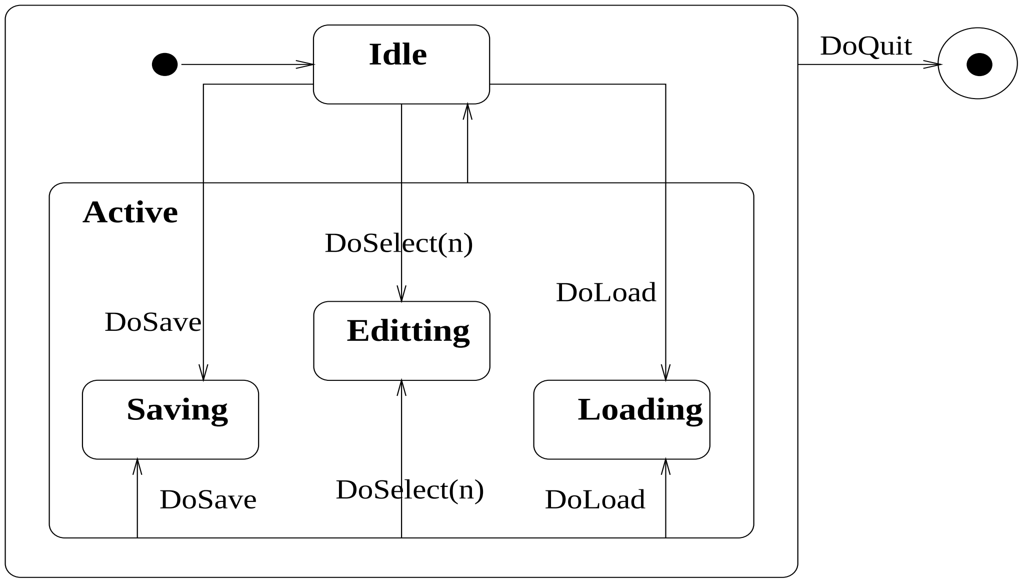

SpreadSheet
....
$create()
$delete()
Clear()
CellContents Get(CellName)
SetCellContents(CellContents)
Set<CellValues> GetChangedCells( )
bool HasArithmeticError()
bool HasCircularity()
String GetErrorMessage()

`$create` and `$delete` are language-dependent constructor and destructor routines respectively

`Set<CellValues>` is instance of *template class* `Set`

is a type representing "set of CellValues"

Dynamic Model



Nested state: `Active`

composed of substates `Editting`, `Saving`, `Loading`

transition can go directly to a substate:
eg `Idle` $\rightarrow$ `Saving`

transition can go from **any** substate by starting at boundary of nested state:

eg `DoSave` from edge of `Active` to `Saving`

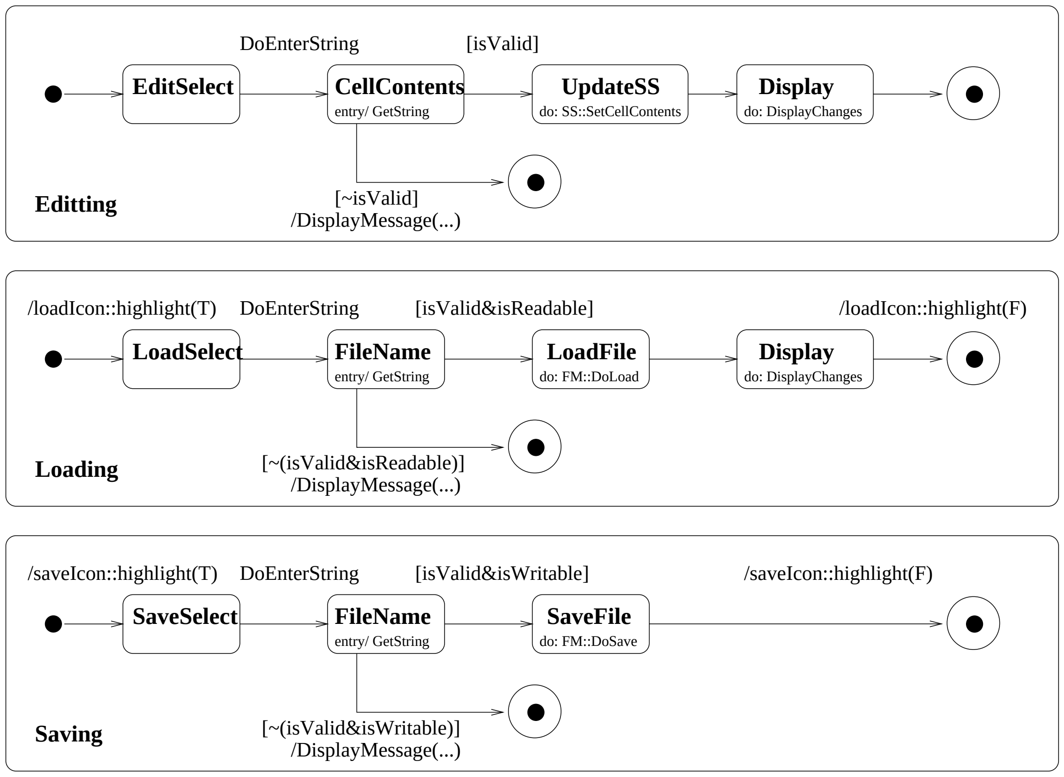
is abbreviation for three (3) transitions, one from each substate of `Active`

Note automatic transition from boundary to `Idle`

when activity of `Editting`, `Saving`, `Loading` terminates, we transition automatically to `Idle`

For next diagram: transition to boundary of nested state is a transition to the initial substate of the nested state

State Diagram (Completed)



transition to boundary of nested state is a transition to the initial substate of the nested state

so `Idle` $\rightarrow$ `Saving` in previous digram

is really a transition `Idle` $\rightarrow$ `SaveSelect`

Entering a final state terminates the “activity” of the substate

so automatic transitions are then able to occur

eg entering one of the above final states allows the automatic transition from boundary of `Active` to `Idle` to occur

Documenting Interfaces

The Architectural Design Document usually specifies interfaces too.

At least for subsystems, and major modules, even if not all class interfaces.

Interface should

- indicate exported types
- specify functions
  - name, arguments, return results
  - pre- and post-conditions
  - exceptions (eg divide-by-zero)
  - informal comments to complement formal spec.
- avoid exporting constant (use function instead)
- avoid exporting variable (use function instead)

Pre-conditions should mention **only**
— arguments
— "state" of module (eg module variables)

Post-conditions should mention **only**
— arguments
— return result
— "state" of module

IS should conform to the standard for design notation

Hints of Specifying Interfaces

When there are several cases,
use several pairs of requires/ensures

```
void Line (Point p, Point q)
  ensures if p and q are in the workspace
      then a line joining them is displayed
      else nothing happens.
```

is better as

```
void Line (Point p, Point q)
  requires p and q are in the workspace
  ensures a line joining p and q is displayed

  requires either p or q is outside the workspace
  ensures nothing
```

In general

```
requires nothing
ensures if P then X else Y
```

can be more clearly expressed in the form

```
requires P
ensures X
requires ¬ P
ensures Y
```