Structure and Patterns in Software Design and the Unified Modelling Language (UML)

Theodore Norvell

Structure

The interrelation or arrangement of parts in a complex entity.

As programs have become more complex, new structuring concepts have evolved to deal with them.

Crucial Book: Structured Programming 1972 contains 3 important essays

- Edsger Dijkstra — On algorithmic structure
- C.A.R. Hoare — On storage structure
- Ole-Johan Dahl & C.A.R. Hoare — On object oriented programming

Quote from last: *... we shall explore certain ways of program structuring and point out their relationship to concept modelling.*

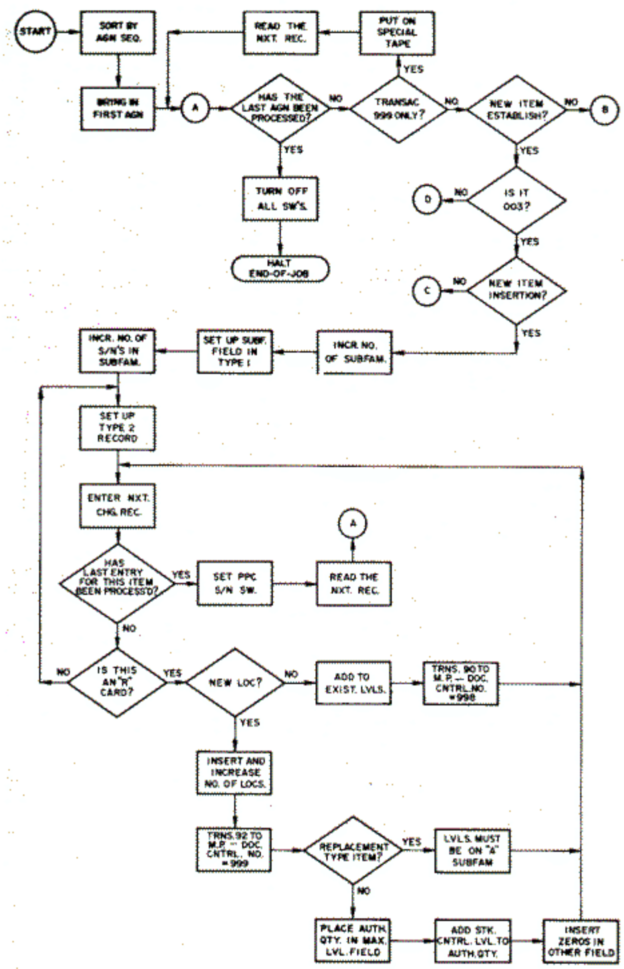
Mainstream structuring concepts

	50s & 60s Prehistory	**70–85 Structured Programming**	**85–01 OO Programming**
Algorithmic structuring	Flowcharts	+Compositional constructs	+Object Interaction
Storage Structuring	Arrays	+Records, unions, pointers	+Object relationships
System structuring	Subroutines	+Modules (packages)	+Templates, Frameworks
Dominant Languages	ASM, Fortran, COBOL	PL/1, Pascal, C, Fortran 77, Ada	Ada, C++, Java
Important Languages	Algol 60, Algol 68, LISP	Simula, Smalltalk, APL, Prolog, Euclid	Haskell, SML

Algorithmic structure — as an example

1950s & 60's: unstructured use of conditional and unconditional branches.

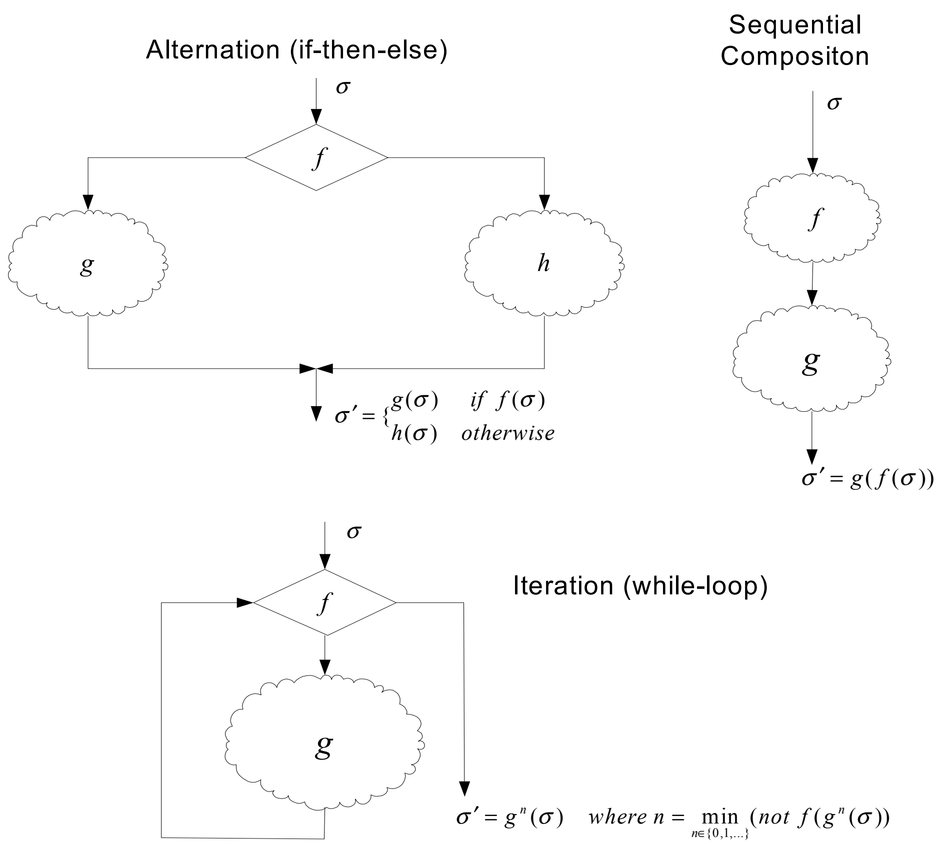
Flowcharts evolved to help S/W engineers visualize the complexity:



Compositional constructs

It was observed that all algorithms could be expressed using only 3 *patterns of composition*

Moreover, each part has a meaning of its own (a function, or more generally a relation)

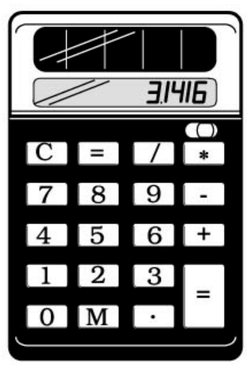


Eventually flow-charts were replaced by pseudo-code, which is *less* expressive, but expresses these patterns well:

if f **then** g **else** h   f;g   **while** f **do** g

System Structuring

- 50s & 60s: unstructured collections of subroutines operating on global data structure
- 70–85: subroutines operating on same data collected in a "**module**" together with that data
  - ∗ Some subroutines comprise the "public interface" to the module
  - ∗ The rest & the data are "private".
  - ∗ Black-box view: Consider a calculator



    · The buttons and the display are the public interface.
    · The algorithms used & the internal registers are "private"
    · We can completely describe the interface without describing the internal working.

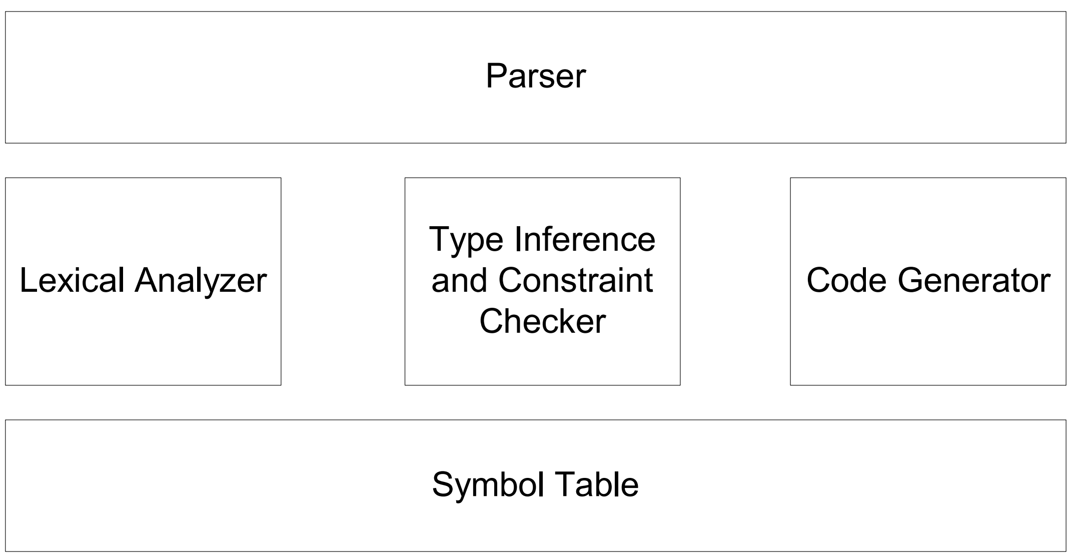
Module Relationships

Modules use (depend on) each other.
Kinds of dependency:

- X calls a subroutine of Y
- X uses a data type defined in Y
- X uses a constant defined in Y

Often dependance is in *layers*. Modules depend on modules below them as bricks depend on bricks below them

A compiler as a layered system



System Structure with Object-Orientation

Each compile-time module has a single run-time instance

$$\text{Static structure} = \text{dynamic structure}$$

With OO systems

$$\textbf{modules} \longrightarrow \textbf{classes} \,\&\, \textbf{objects} \,\&\, \textbf{packages}$$

classes — compile time entities
objects — runtime entities.

- 1 class may have 0 or to run-time instances (objects).
- 1 object belongs to more than one class.

$$\text{Static structure} \neq \text{dynamic structure}$$

Dependence relations between classes are now much more complex

- Each X may call a subroutine of class Y
- Each X may keep a pointer to an instance of Y
- Each X may create a Y
- Each X has a Y as a part of it (*aggregation*)
- Each X is also a Y. (*inheritance* / *generalization*)

Changing Views on Programs

Old view:

- A program is an algorithm that operates on variables

New view:

- A program is a collection of mutually dependant classes.
- A program in execution is an evolving community of relating objects.

The main problem of software engineering is

Mastering complexity.

Errors are more expensive

- if found in design rather than specification/analysis
- if found in implementation rather than design
- if found after deployment rather than in implementation

Thus we must have good notations to master complexity in specification and design.

What is the UML

Premise

Software systems are complex. We need simpler views of them in order to master that complexity.

UML is a language for visual modelling.

- Visual modelling is one way of creating accessible abstractions of complex systems.
- UML is a visual language — follows the tradition of Booch notation and OMT.
- UML supports OO analysis and design.

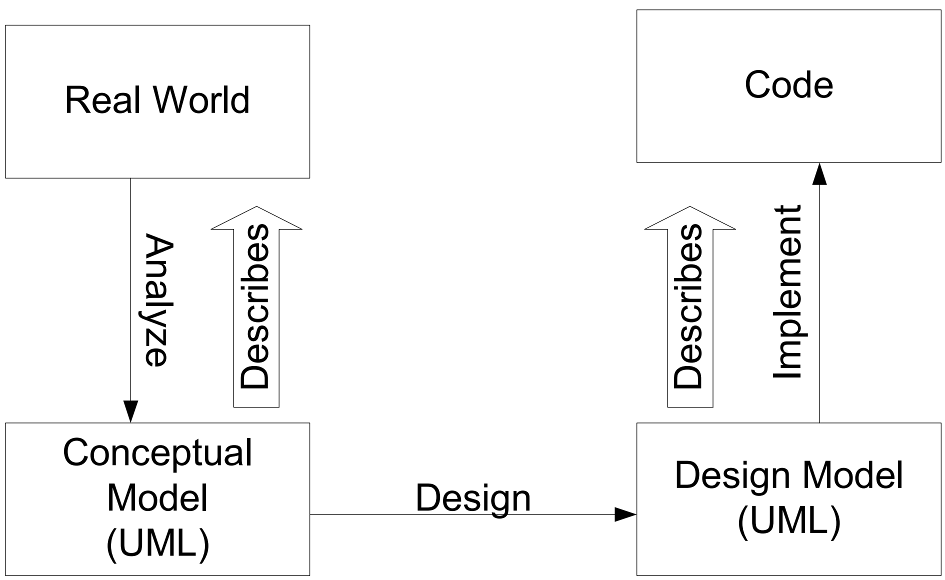
Use of UML

- In analysis and specification phases to model
  - ∗ real-world objects and classes, situations, and processes (e.g., business processes).
  - ∗ existing software components.
  - ∗ interactions between planned software and the above.
- In design phase to model internal components and processes.
- To document legacy systems.

Where UML fits in the software lifecycle

Not only is software complex, it interacts with a complex world. We need abstracted views of the real world (including other software systems).

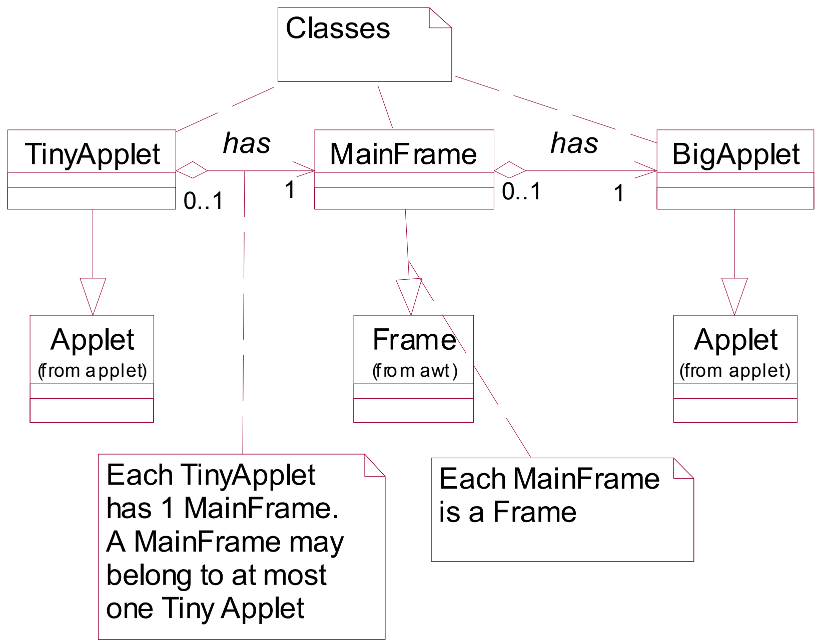
Analysis of the environment has become an important part of requirements engineering and system specification.



Diagrams of UML

- Class diagrams – classes and packages, their properties, relationships.
- Object diagrams – snapshots of objects and their relationships.
- Use-case diagrams – use cases, actors, relationships.
- Sequence diagrams and Collaboration diagrams – typical sequences of events (e.g., calls).
- Statechart diagrams – finite state machines.
- Activity diagrams – algorithms / data-flow.
- Component diagrams – implementation components (e.g. source & object files)
- Deployment diagrams – deployment of components on computers.

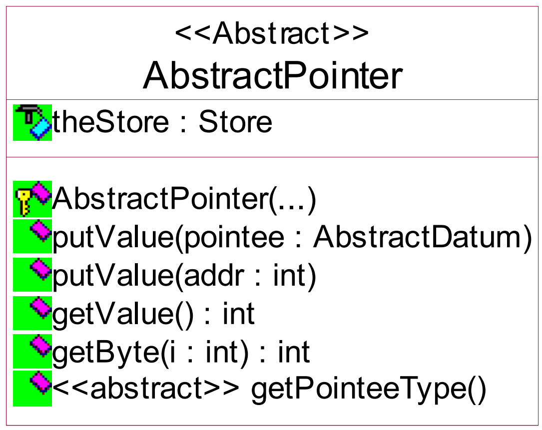
A Class Diagram



Diagrams shows

- 6 classes
- 3 inheritance relationships
- 2 has-a relationships.

Supplying information about a class



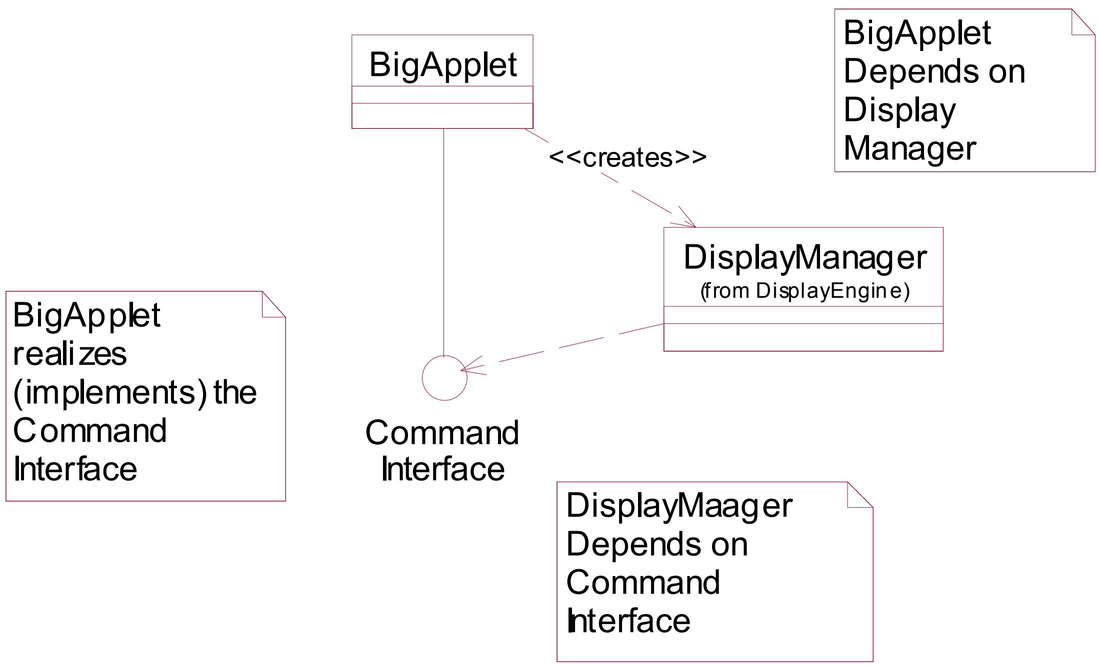
Each class is displayed as a box with 3 or more parts:

- *<<stereotype>> Name*. Stereotypes are used to identify classes that are used in stereotypical ways, e.g. interfaces, abstract classes, actors (agents outside system), exceptions, etc. The Name is the name.
- Attributes. (A.k.a. Fields / data members). This class has one.
- Operations. (A.k.a. Method signatures, function members).
- Other parts as you please. E.g., responsibilities

Operations and attributes are marked according to visibility.

We can model dependance

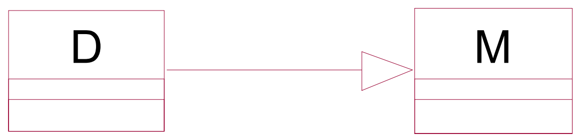
How to do cyclic calling without cyclic dependance.



Class relationships

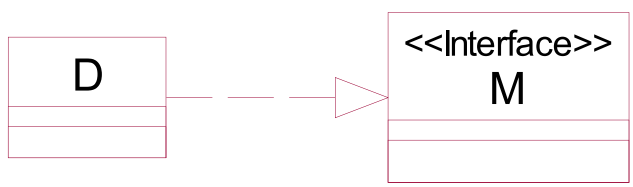
- Is-a (specialization): Every D is an M. Class D specializes class M. Class D inherits from class M.

  In C++ we say D derives from M. In Java D extends M.

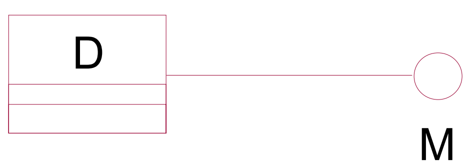


  Note that class D depends on M.

- Realizes. D implements interface M. Special case of above for interfaces.



  or lollypop notation:



- Knows-a (association): Every D can (potentially) easily find an M.

  In C++ (or Java) D might have a data member (field) that is a pointer to an M.

  In the above diagram the D object knows 0 of more M objects. In C++ you might have a data member that is a vector of pointers to M objects.

  Use a two way arrow if the M object can find the D object that can find it.

  Use no arrow if there is an association, but you don't want to imply that either can find the other.

  Usually (with the arrow) D depends on M.

- Has-a (aggregation): Every D has an M's.

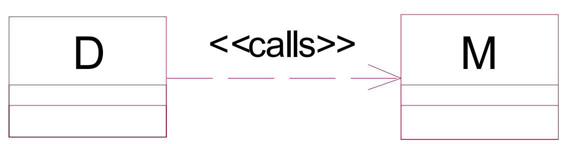
  This is a special case of "knows-a". Use it when the lifetimes are coincident; i.e. creating a D object creates the M object and destroying the D object destroys the M object.

In C++, D might have a private data member of type M called name, or D might have a pointer to an M object that is set with new when a D is constructed and sent to delete when a D is destructed..

- Depends on: Use when there is dependance, but none of the above are appropriate.

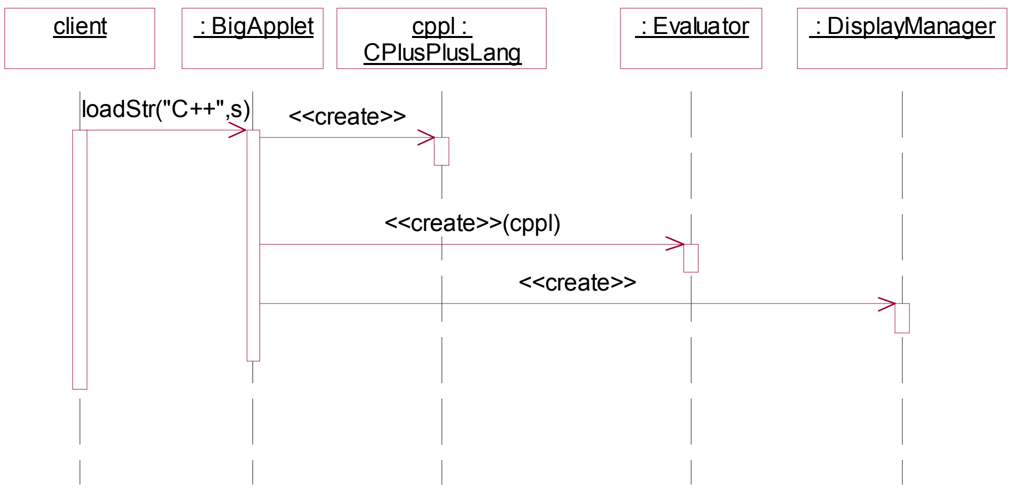
  E.g. Some method D.foo() takes an M as a parameter, returns an M as a result, creates an M, but doesn't maintain a long term association, or calls a static method of M.

It is good to use a stereotype to describe the type of dependance. E.g.:

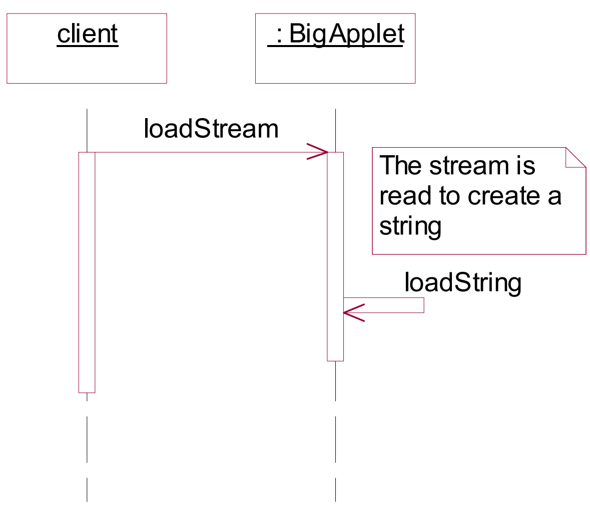


Sequence diagrams

Show typical scenarios – not algorithms.

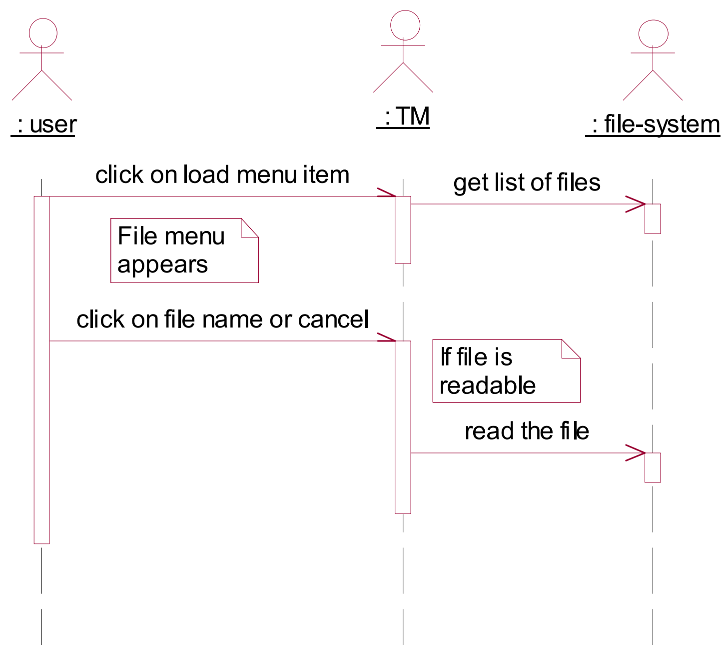


Messages may be sent to self



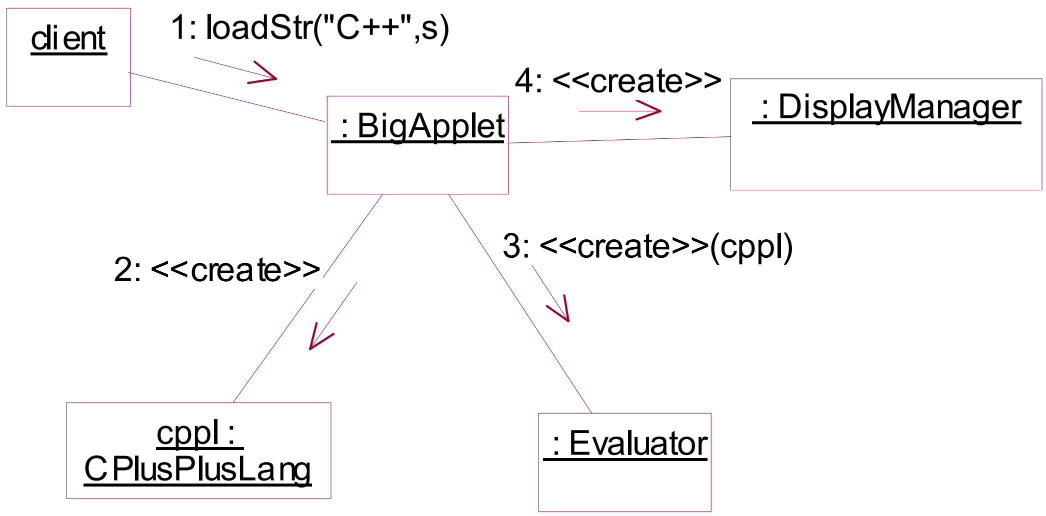
Sequence diagrams

... can show the interaction of a system with objects outside (specification)



Collaboration Diagrams

Same info as sequence diagram, but in different form

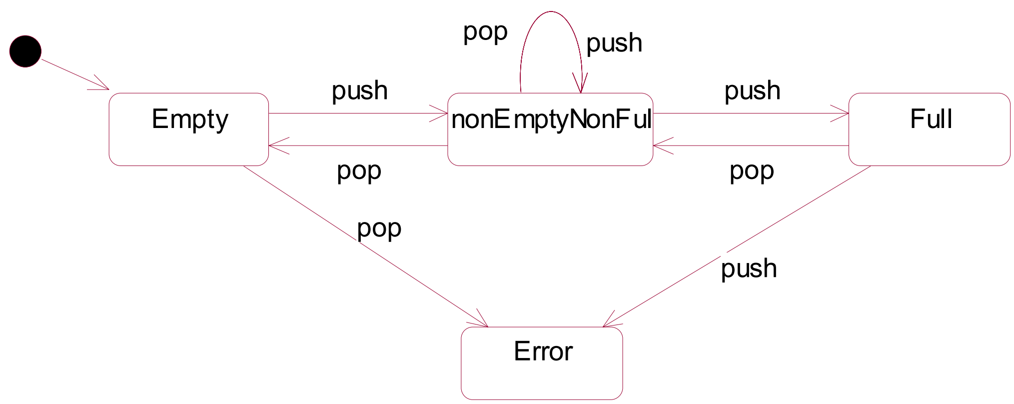


State Diagrams

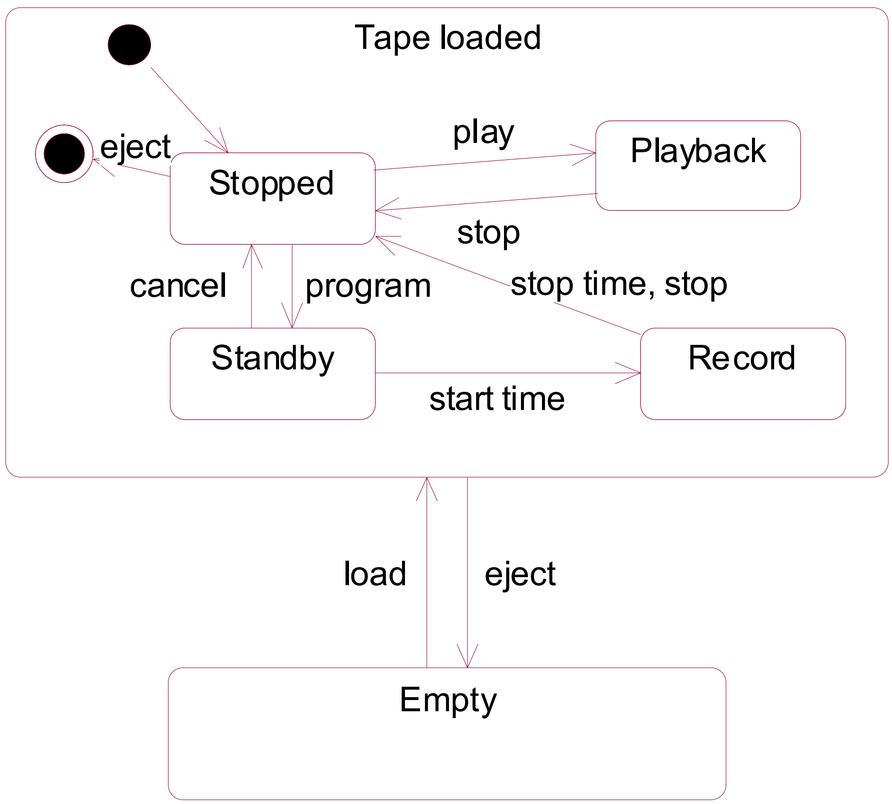
Describes the semantics of an interface in term of the states of the objects.

Usually abstracts related states to a single state.

A stack.



A VCR showing substates



Patterns

Two stories

Romeo and Juliet | Jack and Rose

meet but they are divided by

family fueds | class and betrothal

Nevertheless, they fall in love.

All is bliss until

Tybalt dies | the ship hits an iceberg

culminating in the death of

both. | Jack.

Similarity both of characters’s relationships and of plot.

The star-crossed lovers pattern

Boy and girl

meet but they are divided by

social circumstances.

Nevertheless, they fall in love.

All is bliss until *fate intervenes*

culminating in their separation by death.

Patterns of OO Design

Use of flow charts led to the realization that

- Hierarchical decomposition is good
  - ∗ parts have meaning, parts have parts with meaning
- A small number of composition rules suffice
  - ∗ and are worth naming and paying attention to

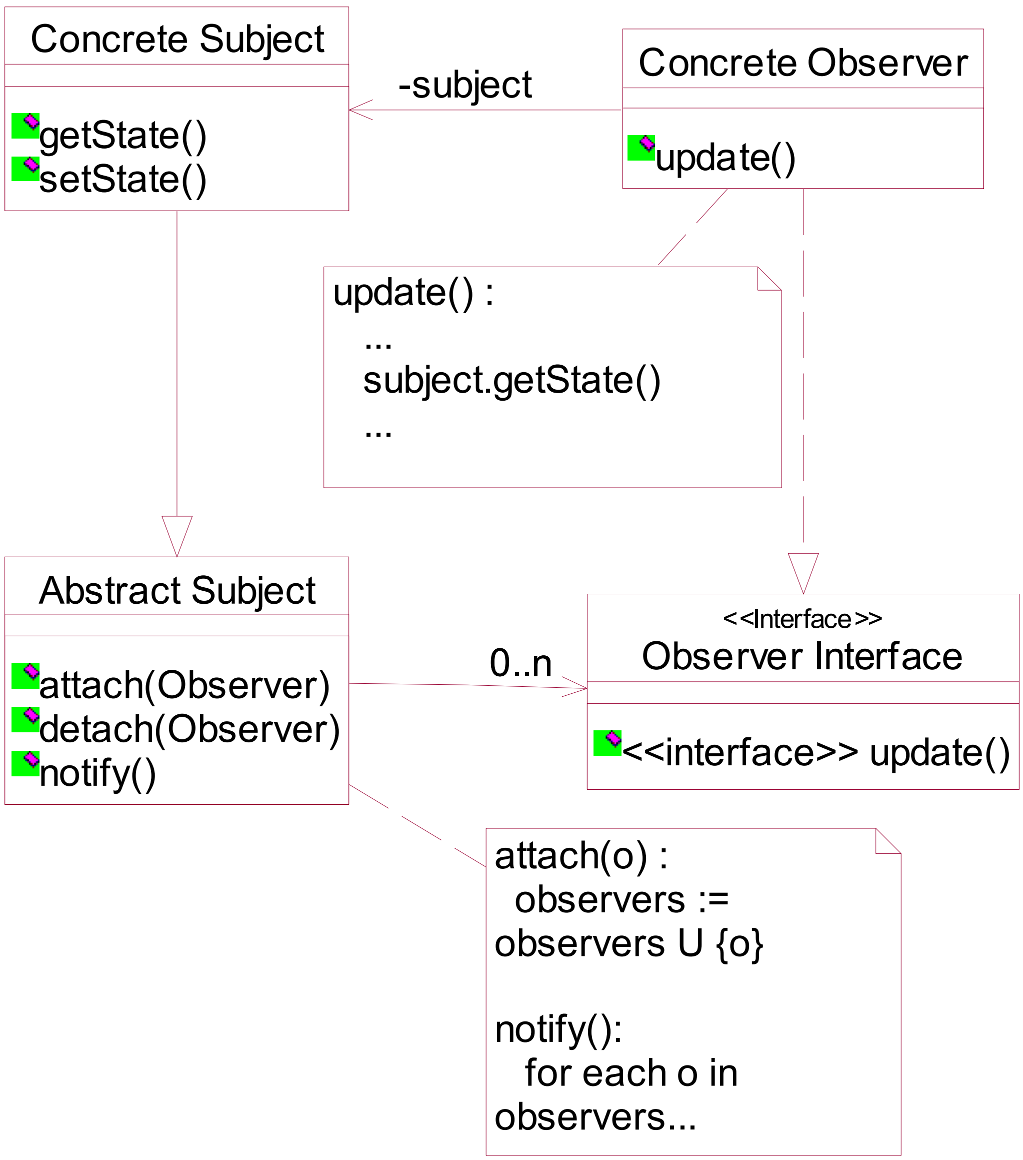
A second important book

Design Patterns by Gamma, Helm, Johnson, Vlissides 1995

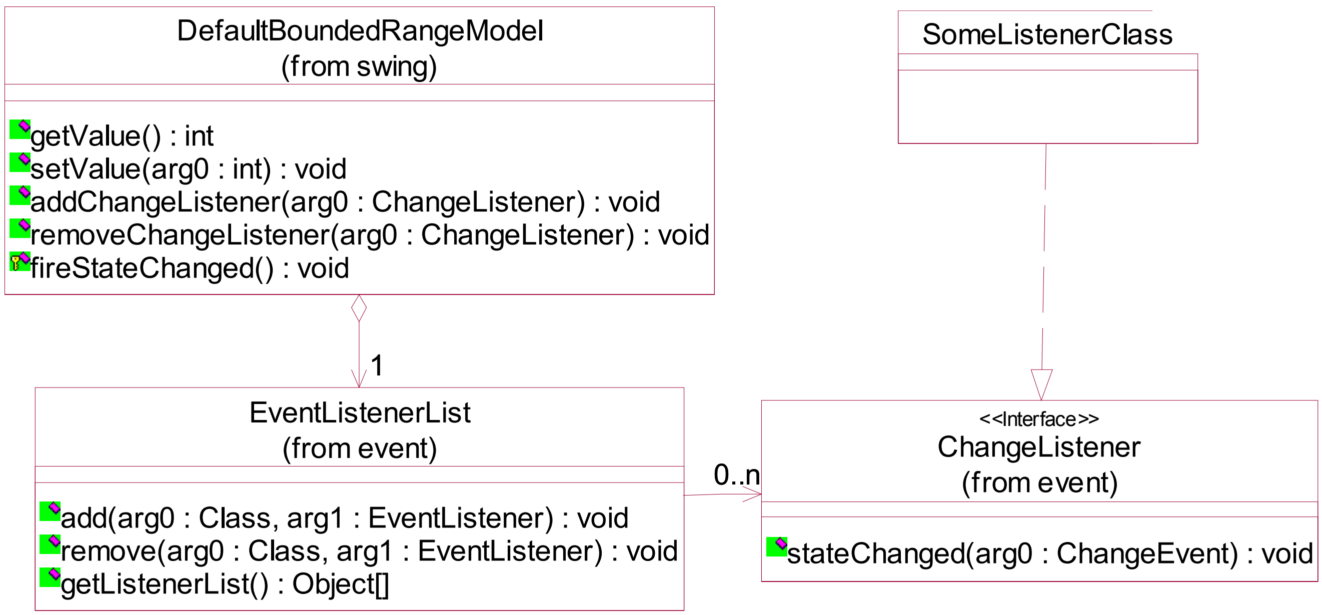
Explores patterns of object composition that recur and lead to "good" designs.

The Observer Pattern

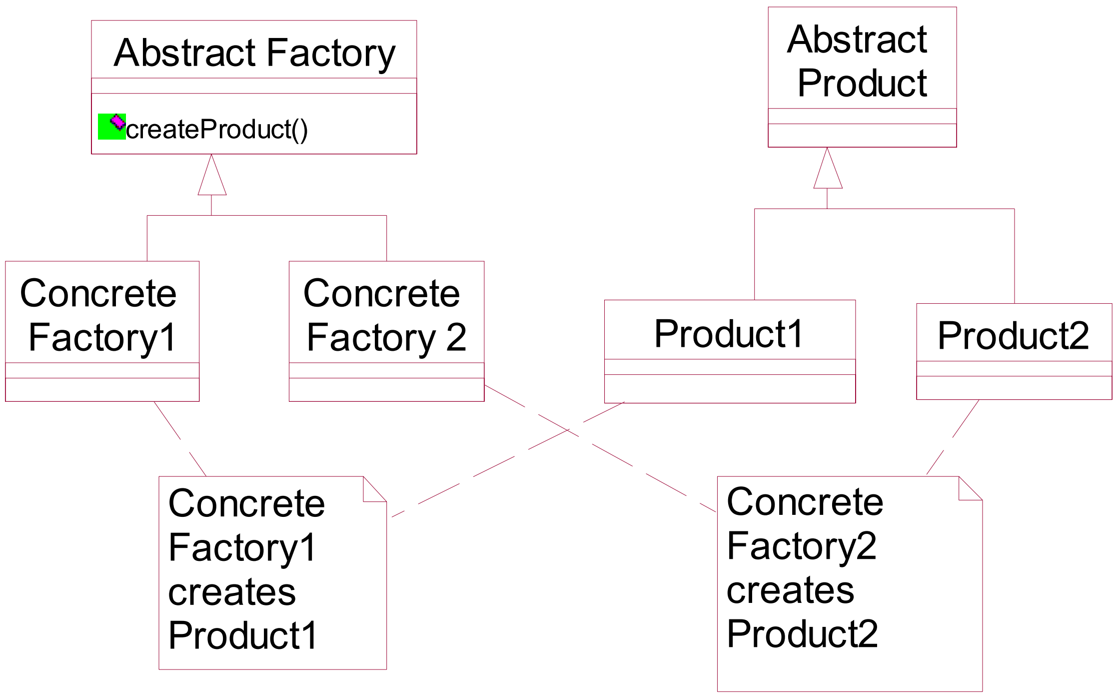
From Gamma et al.



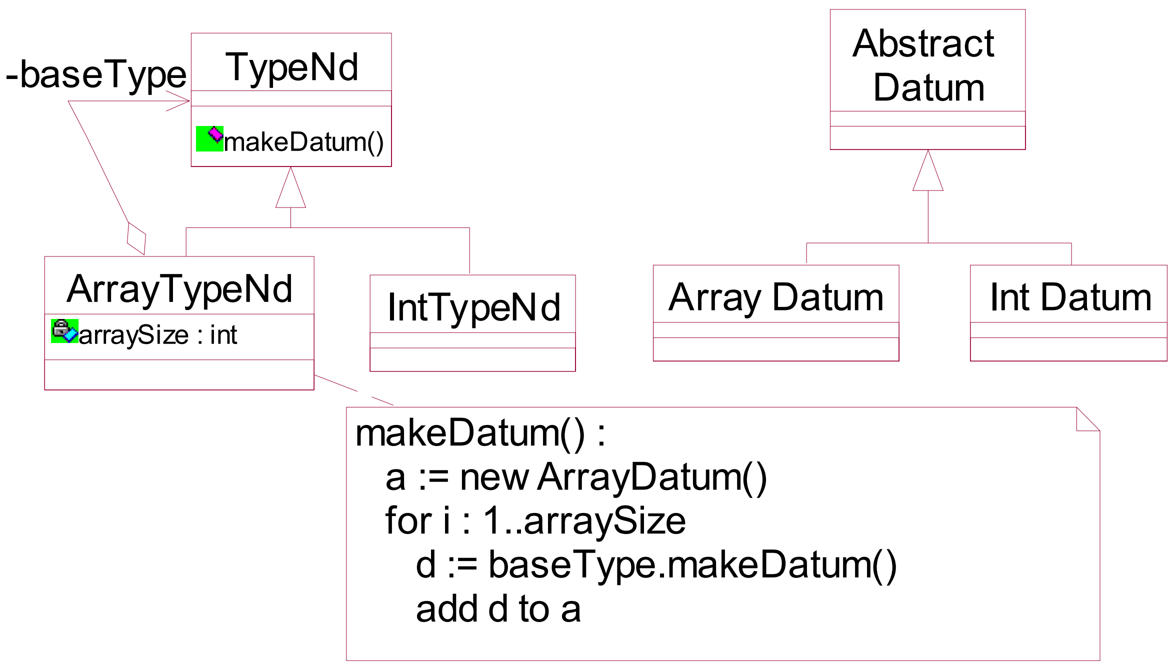
The Observer in SWING



Abstract Factory Pattern



Abstract Factory in the Teaching Machine



Patterns as a structuring device

Gamma et al describe 23 patterns of the following kinds

- *Behavioural patterns* — include interactions across time
- *Structural patterns* — how objects aggregate
- *Creational patterns* — solve problems with object creation

Patterns give the S/W engineer a new vocabulary with which to understand software systems and to create understandable designs.

Structure Patterns and Structuralism

Structuralism in Linguistics, Sociology and Anthropology
Claude Levi-Strauss is major figure.

- Focuses on structures on mind and society.
- Teaches that not all structures are equal in the human mind.
- May have much to teach software engineering
- May have something to learn from software engineering.

Conclusions and Assessment on UML

UML has considerable momentum.

- Lots of books.
- Good industry uptake.

UML is big and expandable.

- It offers something to everyone.
- But it is weak on data flow.
- Assertion language (OCL) is defined, but not widely known and may define semantics of classes better than state or activity diagrams.

Tools

There are several tools that hold models

- Keep diagrams consistent with database.
- Automatic analysis of source code.
- Automatic generation of source code.
- Round-trip engineering.

Conclusions on structuring

History shows frequent paradigm shifts in software engineering.
The story is not over.
New structuring ideas will appear.
Those not open to them will be left behind.
Notations help abstract away from complexity.

- It helps to look at the complex in simple ways.

Eliminating complexity is better than abstracting it.
Mastering complex systems is not limited to software engineering.
It also applies to

- Computer Engineering in general.
- Electrical Engineering & Systems Engineering