Software Design

Software Design

- According to the Waterfall model, Design should be undertaken for the whole system in one shot.
- In most cases, this is impractical, rather use an incremental way of designing.
  - Design the core or architecture of the system
  - While (system incomplete) do
    - Design one part completely
    - Code and test that part
- Permits a better use of resources if needed
- Risk: degradation of system structure through iterations
- Overhead required to make each part run separately.

Conceptual Design

- The conceptual design answers questions such as these:
  - Where will the data come from?
  - What will happen to the data in the system?
  - What will the system look like to users?
  - What choices will be offered to users?
  - What is the timing of events?
  - What will the reports and screens look like?
- A good conceptual design should have the following characteristics:
  - It is written in the client's language.
  - It contains no technical jargon.
  - It describes the functions of the system.
  - It is independent of implementation.
  - It is linked with the requirements documents.

Conceptual Design

- Problem: there doesn't seem to be much difference between the conceptual design and the requirements! The difference is this:
- The requirements describe *what is required*.
- The conceptual design describes *what will be provided*.
- Here is a simple example:
  - Requirement: the user must be able to open a file.
  - Conceptual design: the user opens a file by performing the following actions.
    - Select **File** from the main menu. System displays a list of files.
    - Select a file.
    - Select **OK** or **CANCEL**.
- Both can be used as contracts between supplier and client.

Technical Design

- Description of the hardware and software components and their functions
- Description of data structures and data flow.

  - Usually shows how the conceptual design can be implemented by a collection of *components*.
  - Each component has an *interface*.
  - Components *interact* through their interfaces.

Wasserman's Classification

- Wasserman suggested that designs are created in one of five ways.
- Each way has a *high-level description* and a *low-level description*.

Wasserman's Classification

- Modular Decomposition. Assign *functions* to *components*.
  - High-level description: functions to be implemented.
  - Low-level description: how component works and relates to other components.
- Data-oriented Decomposition. Organize design around principal *data structures*.
  - High-level description: general data structures.
  - Low-level description: what data elements store and how they are related.
- Event-oriented Decomposition. Organize design around the *events* that the system must handle.
  - High-level description: states and events.
  - Low-level description: how state transitions occur.

Wasserman's Classification

- Outside-in Design. Organize the design around user inputs, treating the system as a "black box".
  - High-level description: a list of all user inputs.
  - Low-level description: responses of system to inputs.

- Object-oriented Design. Organize the design around *objects* and the relations between them.
  - High-level description: list of classes and the relationships.
  - Low-level description: attributes and methods of each class.

Wasserman's Classification

- Comments on Wasserman's classification:
  - The categories overlap: an actual design could match several of them.
  - Different strategies are useful for different kinds of problems.
  - Modular decomposition and outside-in design are ***not recommended***. They are both based on top-down techniques that are fairly effective for small applications but do not work well for large applications.
  - Data-oriented design has been largely subsumed by object-oriented design.
  - Event-oriented design should usually be combined with something else --- typically, object oriented design.
  - Object oriented design is currently dominant.

Software Architectural Design

Software Architecture

- If we think in terms of "building software", the idea that a software product has an "architecture" is natural.
- "Architecture" is just another name for the overall organization of the software.
- The importance of software architecture, and the number of varieties of software architecture, have been realized and exploited only recently.

- Architecture styles are most often combined.

- Some architectural styles are:

Hierarchical Architecture

- A hierarchical architecture has the form of a tree in which the root is the "main program" and the leaves are primitive functions (that is, functions that do not call other functions). Intermediate nodes of the tree are functions that call and are called by other functions.
- Hierarchical organization is usually the result of top-down design: a problem is recursively decomposed into smaller and smaller parts. Hierarchies have several disadvantages:
  - they are designed to perform a single function (the "main program");
  - they are organized around control flow (functions calling functions) --- data tends to be neglected;
  - the "leaf" functions are very specialized and usually cannot be used in any other applications.

Layered Architecture

- Components arranged in layers.
  - The lowest layers perform the simplest, most concrete tasks
  - Higher layers perform progressively more complex and abstract tasks, using the lower level(s) functionalities.
  - The top layer is often effectively the user interface of the system.
- Generalization of the hierarchy.
  - Most actual hierarchies are actually layered because leaf nodes are shared by higher-level nodes.
- A higher layer may use the services of its own layer and lower layers but a lower layer may not use the services of a higher layer.
  - In some systems, there is a stronger constraint: a layer can use services only of the layer just below it
- Operating systems are often constructed in layers.
- Sometimes described as rings, e.g. MULTICS.

Pipes and Filters Architecture

- **Pipe:** Channel that connects two software components; one component produces data and the other consumes the data.
- **Filter:** Component that inputs data from one or more places, processes the data in some way, and outputs it to other places.
- Systems are built by using pipes as connectors between components.
- Example: UNIX pipes: $ who | pr -3 | lpr
- Dataflow approach

Event-Oriented Architecture

- Traditional architectures assume that the flow of control is determined by the programmer.
- Event-oriented architectures assume that any event can occur at any time.
- They are often separated into a layer that is sensitive to events and a layer that processes the events.
- The processing layer must provide the event-handling layer with pointers to a set of functions called *callback functions*.
- When an even occurs, the event handling layer detects it, selects the appropriate callback function, and invokes it.
- Example: window system

Repository Architecture

- System organized around a collection of data that is shared by many processes.
- Example: Some software development environments (SDEs) are organized around a repository that holds source code, compiled code, executables, symbol tables, documentation, and other artifacts related to software development. The SDE provides a set of tools --- editor, compiler, debugger, revision control, documentation manager, etc. --- that examine and update the data in the repository.

Client-Server Architecture

- A **client system** issues a request that is handled by a **server system** that provides services to its clients
- Suitable for most transaction-oriented systems.
- Example: The server might provide a simple service, such as storing files for a community of users, or a more complex service, such as booking airlines seats for customers.

“Object-Oriented Architecture”

- The object oriented paradigm is sometimes referred to as an “architecture”.
- Object-orientation is a paradigm, and is really more general than that.
- Most architectures (including those listed here) can be implemented in an object oriented way.

Software Design Documentation

Design Documentation

- A typical set of design documents includes:
- The Conceptual Design: description, in non-technical language, of the ways in which the system will meet the client's requirements.
- The Technical Design: a document in two parts.
  - The System Architecture (also known as the High Level Design):
    - description of each component of the system
    - diagrams showing how the components are related
  - The Detailed Design:
    - a description of each component. (data structures, functions)
    - enough detail to enable a programmer to write code
- Should include the rationale of each design elements

Good Design

Qualities and Techniques to Achieve Them

Good Design

- What is a good building?
- How do we create a good building?

- How do we tell if a design is good or bad?
- How do we create a good design?

- Not easy to answer, even worse for software.

- First step is a good architecture.
- Then, try to arrange thing orderly.

Design Qualities

Modularity

- A good design usually consists of a collection of well-defined, discrete components or modules.
- There are various ways of organizing the modules (software architecture)
- The important thing is to avoid a jumble of classes connected in arbitrary ways.
- Several levels of abstraction in modularity.

Cohesion

- A module is *cohesive*, or has *high cohesion*, if its internal parts are closely related towards the achievement of a clear purpose.
- Every sub-component is working towards this goal.

- A simple test of cohesion is to try to describe the module concisely.
- There are several levels of cohesion

Cohesion

- Coincidental:
  - parts are unrelated to one another
- Logical:
  - logically related functions or data elements are placed in the same component. E.g. all “input” features are put in the same component, either from files, or from the network.
- Temporal:
  - functions that are executed in sequence are put in a component. E.g. system initialization procedures.
- Procedural:
  - functions that sequentially aim at the production of a result are put in the same component. E.g. capture the data, validate, create a record, and save it.

Cohesion

- Communicational:
  - functions are associated because they operate on or produce the same data set. E.g. book inventory is used for both accounting and managing orders.
- Sequential:
  - the output from one part of a component is input to the next part. E.g. compilers.
- Functional:
  - every processing element is essential to the performance of a single functionality, and all essential elements are contained in one component.
  - A functionally cohesive component not only performs the functionality for which it is designed, but also performs only that functionality and nothing else.
  - It is thus more likely that changing this particular functionality will affect only one component.

Coupling

- Two modules are *coupled* if they depend on each other in any way.
- As with cohesion, there are various degrees of coupling.
- Unlike cohesion, less coupling is better.
- Is it possible to have NO coupling at all?

- **High cohesion and low coupling** is a great goal to reach

- Example: façade pattern

Coupling

- Content :
  - one component actually modifies another. E.g. one component modifies an internal data item in another component, or when one component branches into the middle of another component.
- Common :
  - data are accessible from a central store in both components. E.g. global variables. it can be difficult to determine which component is responsible for having set a variable to a particular value.
- Control :
  - one component passes parameters to control the execution of another component. E.g. setting a “flag”. Sometimes acceptable, but should minimize the amount of controlling information that must be passed from one component to another and to localize control to a fixed and recognizable set of parameters forming a well-defined interface.

Coupling

- Stamp :
  - a data structure is used to pass information from one component to another, and the whole data structure is passed. Bad if some of the data is not used.
- Data :
  - a restriction of stamp coupling, where data is passed to the called component, but where only the necessary information is passed. Ideally, all informations are passed as separate parameters, even though they come from the same data structure.
- Uncoupled :
  - no interconnections at all. Possible?

Information Hiding

- In a famous paper written in 1972, David Parnas set out the principles of *information hiding*:
  - *"The developer of a software component must be provided with all the information necessary to carry out the given responsibilities assigned to the component, and should be provided with no other information."*
- Highly related to cohesion and coupling, but sufficiently important to be separated from them.
- If designers do not follow the principles of information hiding, the system will contain *hidden dependencies* that will make maintenance a nightmare.

Fault Tolerance

- A good design should be tolerant of both external and internal errors.
- It is important but straightforward to protect the system against most external errors. The key step is to validate all inputs to the system carefully, so that the system rejects inappropriate data and processes only "good" data.
- Even after validation, internal errors may still cause the system to fail. Dividing by zero or computing the square root of a negative number causes most processes to raise a signal. If the signal is not handled and processed correctly, the system may crash.

Simplicity

- Always choose the simplest design when a choice comes in. (if both designs meet the requirements)
- Designer should try to remove all *unnecessary* complexity from the system.
- Simplifying is hard and requires experience.
- Examples:
  - a design diagram is hard to read if it contains many crossing lines. If you redraw it to reduce the number of intersections, you will understand it better and may be able to simplify it.
  - A large number of links to a class may indicate low cohesion or high coupling. Increasing cohesion and reducing coupling will tend to simplify a design.
- Don't worry about efficiency during design.

Tips & Techniques to Achieve Design Quality

Prototyping

- If you are not sure whether a design will actually work, it may be a good idea to *prototype* it.
- This means implementing the design in a “quick and dirty” way.
- Don't invest a lot of time in programming, but get something running that is just enough to validate the design.
- Be very careful if you reuse the prototype code in the operational version.

Dividing Responsibilities

- A useful principle in object oriented design is that responsibilities should be divided evenly between classes.
- A system in which one class does all the work and the other classes merely store data is probably badly designed.

Design Rationale

- During the design process, the designers discuss many alternatives and reject all but one.
- It is important to include the reasoning in the design so that maintainers do not waste time implementing the rejected alternatives in the hope of improving the system.

Efficiency & Design

- Design requires compromising. There are many trade-offs to be made during design
- A good designer is a person who has a good feeling for what is important and what is not.
- A common mistake is to place too much emphasis on efficiency, which is not ever-present:
  - What are the time-critical aspects of the system?
  - How can we ensure that the critical time constraints are satisfied?
- When we have answered these questions, we can forget about efficiency and concentrate on other aspects of good design.
- A useful rule of thumb is that 90% of the time is spent executing 10% of the code.

Sum-Up: Steps to Good Design

- **Increments.** Start with any design, and improve it.
- **Information Hiding.** For each class, ask: what information does it hide? Should it hide more or less information?
- **Coupling**. Which classes are highly-coupled (i.e., have many dependencies with other classes)? Is the coupling necessary? How can it be reduced?
- **Cohesion**. Can each module be describe in one sentence? Does each class have a cohesive set of functions? Is every function needed? Are more functions needed?
- **Trade-offs**. Balance efficiency against abstraction, simplicity, and coupling --- assuming that in most situations, efficiency is less important than the other factors.

Design Reviews

Design Reviews

- Reviewing is an important part of software engineering (and engineering in general).
- Any product of development can be reviewed:
  - requirements, specification, design, implementation, test results, etc.
- Reviewing is generally performed by a ***review team***
  - people responsible for the product
  - others who can study the product with fresh eyes.
  - including the authors might not be desirable
- It is important that reviewing is an "egoless" activity.
- A reviewing team does not usually correct the errors that its members find. Instead, the views of the team are recorded accurately.

Design Reviews

- The **preliminary design review** :
  - meeting of clients and suppliers
  - the conceptual design is reviewed and approved by the clients.
- The **critical design review** :
  - meeting of designers.
  - requirements team (sometimes called “analysts”) may also be present
  - ensure that the conceptual and technical designs are free of defects and meet the requirements
- The **program design review** :
  - meeting of designers and developers.
  - ensure that the detailed design is feasible
  - the implementation team will be able to understand it.

Design Reviews

- All of these reviews are important.
  - In a small project, however, the first two would be fairly brief and informal.
  - The program design review is the most important and should be carried out carefully, even for a small project.
- Reviews are useful: consider including reviews in your project.