Looking at Design

Overview

- Motivation
- Design in general
- Software design
- Conclusions

Motivation

- Is there a ‘Software Crisis’?
- Many software projects are unsatisfactory
  - lots fail to meet their design goals
  - lots exceed budget or time constraints significantly
  - some are total disasters and are abandoned at huge cost
  - see Computer Weekly for regular examples of software disasters
    - often paid for by the taxpayer!
    - e.g. air traffic control, health software, passport office
- Many reasons for software project failure
  - but good software design is a critical weapon against such problems

Motivation

- Good software design
  - reduced uncertainty
  - improved quality and predictability
  - improved chance of meeting the design goals
  - improved chance of finishing on time and within budget
  - documented history of what you were trying to achieve
- Makes your life better
  - less time debugging
  - adding new features will be easier
  - less stress, fewer grey hairs
  - more reward

Motivation

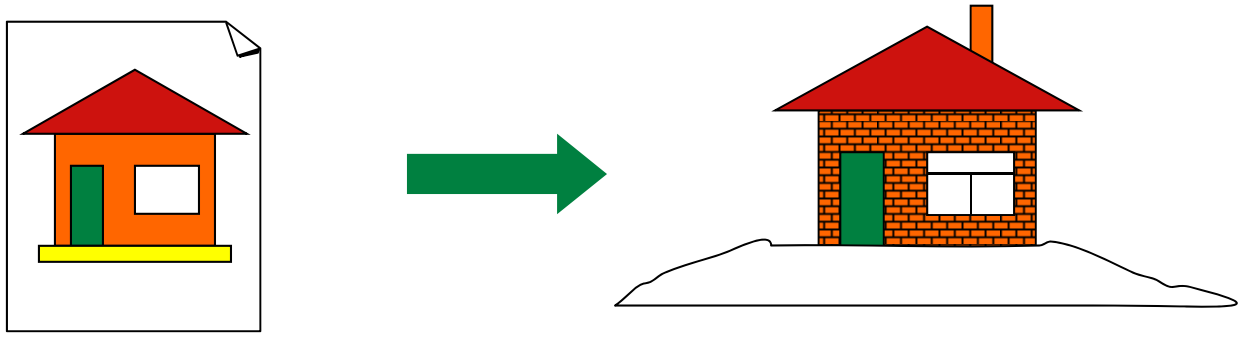
- People will like you
  - whoever's maintaining the software will thank you
  - customers will respect you and come back for more
  - bosses will promote you

Introduction to Design

- What is design?
  - ‘the act of working out the form of something, as by making a sketch or outline or plan’ Wordnet
  - in this course the something is referred to as an *item*
- What is *a* design?
  - it’s the sketch or outline or plan for the item
  - it’s a description of some kind
    - or indeed the final item itself
  - it’s an approximation to the form of an item

A Design

- Here's a design and a final item
  - note some differences between the design and the final item
  - the design is an approximation to the form of the final item



design

final item

What's the Point?

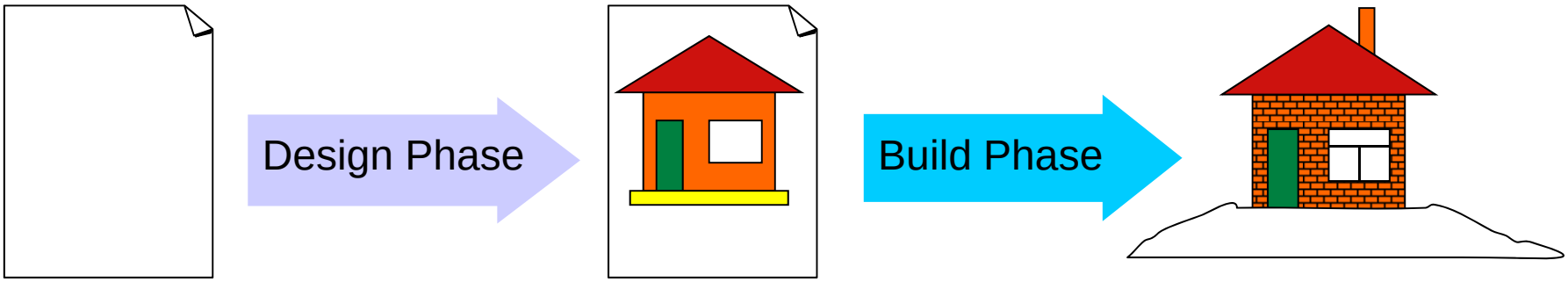
- The point of a final item is to solve some problem for humans
  - a car solves the problem of getting to the shops
  - a jumper solves the problem of being cold
- But an item only solves the problem if it satisfies certain goals
  - a car doesn't solve the problem if it's too small to get inside
  - a jumper doesn't solve the problem if it's too thin
- The point of design is to improve the chance of the final item meeting the design goals

Design Goals

- Functional goals
  - ‘what it does’
  - e.g. the item must transport at least one person
  - e.g. the item must allow someone to stay warm in winter
- Performance goals
  - ‘how well it does it’
  - e.g. the item must have a top speed of at least 30 mph
  - e.g. the item must not be heavier than 0.25 kg
- A ‘good’ final item is one which satisfies the design goals

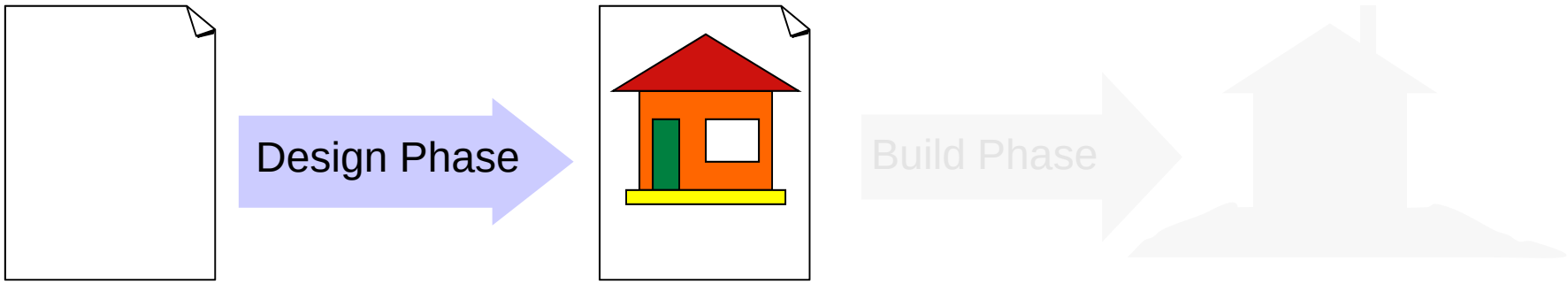
A Project

- The aim of a project is to arrive at a good final item
- How do we get from ‘no design’ to ‘final item’?
  - by a series of steps
  - traditional projects often divide into two main phases



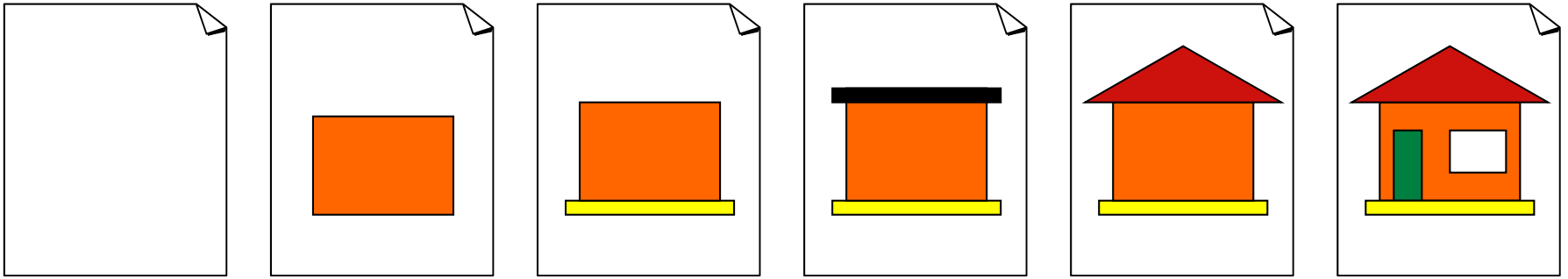
The Design Phase

- How do we get from ‘no design’ to ‘last design’?



Design Evolution

- Example: design something to live in



no design	add walls to keep wind out but will subside	so add foundations but rain will come in	so add flat roof but water will collect	so change to sloping roof but can't get in	so add door

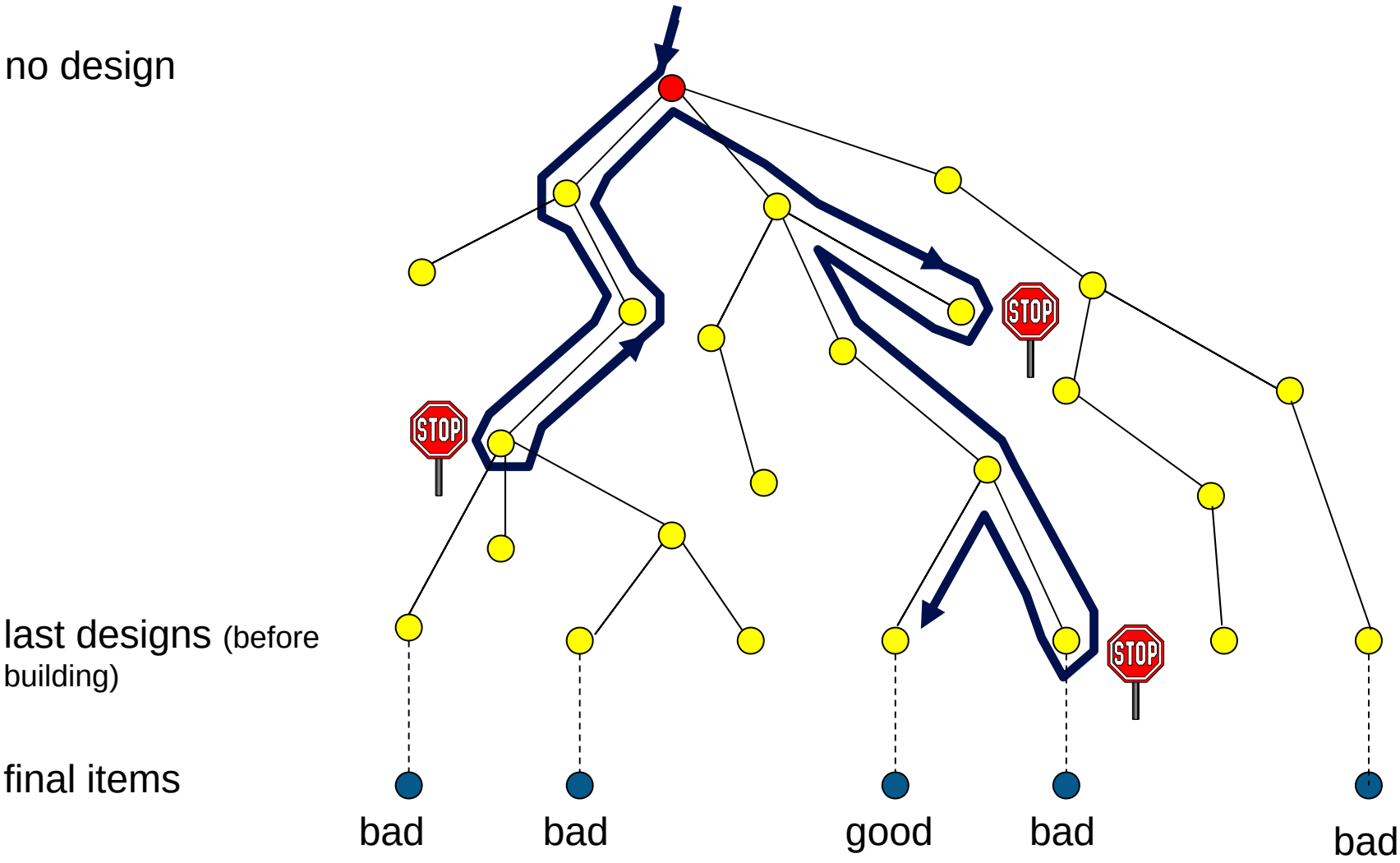
Design Evolution

- Designing involves two main things:
  - 1. having ideas
  - 2. realising they're rubbish
- Identify and correct design defects
  - which will otherwise lead to the final item failing the design goals
  - problems, errors, inconsistencies
- It's an inventive, difficult process
  - have to generate lots of ideas
  - being full of ideas isn't easy!

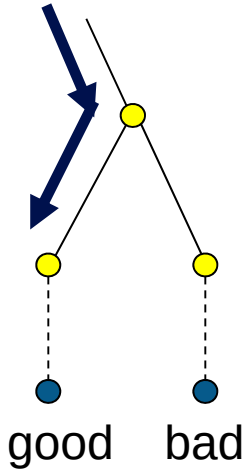
Design Evolution

- It's an uncertain, unpredictable process
  - don't know if or when a design defect will be identified
  - don't know how much back-tracking will be required
  - very difficult to estimate how long it will take to design something
- It's 'Iterative Refinement'
  - may involve significant back-tracking
  - it's rather like exploring a tree (in computer science)

The Design Tree



Design Evaluation



- Need to somehow find a path that leads to a good final item
  - otherwise the project will be a failure
- Want to minimise the length of the path
  - can't afford to search exhaustively down every branch
  - some back-tracking is inevitable but we want to minimise it
- So for any design we need to evaluate whether it's on a good or bad path
  - in particular, for the last design we need to predict how good the final item will be before we build it

Design Evaluation

- Key problem: design evaluation is very difficult
  - search for design defects, inconsistencies etc.
  - if you find them then the design can't be very good so correct them
  - but still can't guarantee that the final item will satisfy all the design goals until you actually build it
- No easy answer
  - no substitute for experience and practice
  - but you can use the 'Big 3 Design Criteria' to help gain insights into how good your design might be

The Big 3 Design Criteria

- 1. Detail
  - how approximate is the design?
- 2. Intersection
  - how much common ground is there between the design and a good final item?
- 3. Merit
  - how many desirable properties does the design have?

1. Detail

- How much detail does the design have?
  - how closely does it approximate a final item?
  - how far down the design tree is it?
- Designs can differ widely in how closely they approximate the final item
  - e.g. a scribble on the back of an envelope
  - e.g. a 500 page design document
  - both are designs but at different levels of detail

High Detail

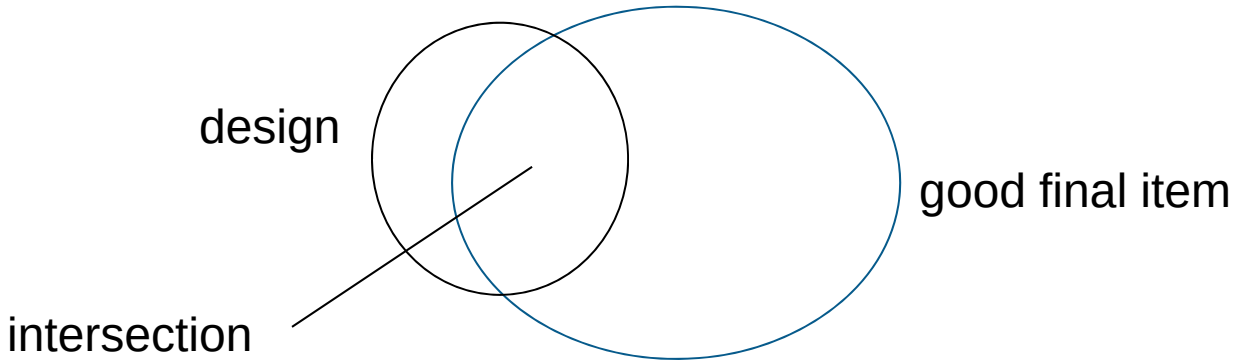
- A design may be a close approximation to an item
  - a house architect's blueprints
  - a well-written cake recipe
  - a silicon-level layout schematic for a new computer chip
  - the final item itself
- All have a high level of detail
- Greater understanding of the final item
- Less risk and uncertainty

Low Detail

- Many designs are not such good approximations
  - significant absences in the design
- Low detail designs
  - haven’t solved enough of the problem
  - are insufficiently specified to allow predictions about the final item
  - the only valid prediction is that there is more design work to do

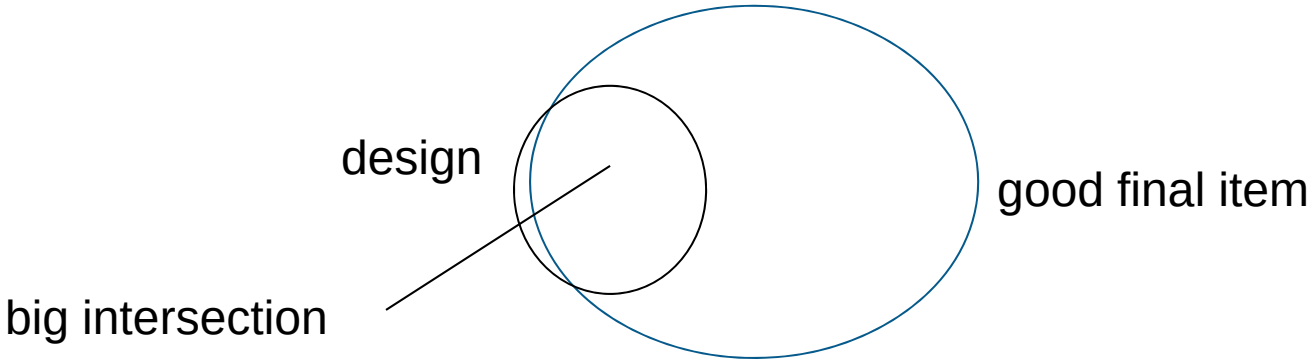
2. Intersection

- Intersection is the common ground between the design and a good final item
  - the amount of key features that are common



High Intersection

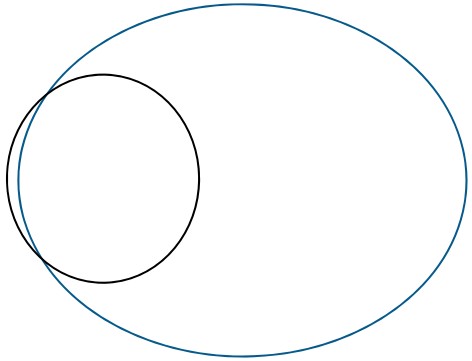
- High intersection is very valuable
- Even a low detail design may be quite useful if it has high intersection
  - because it's something you can build on
  - think of such a design as high up the design tree but on a good path



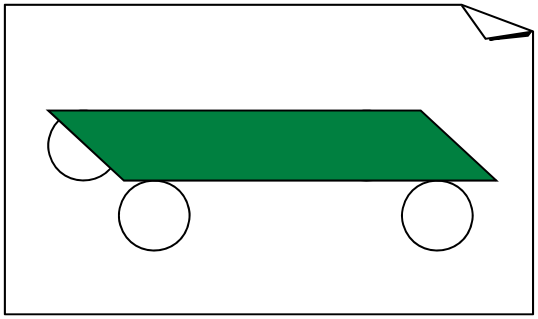
High Intersection

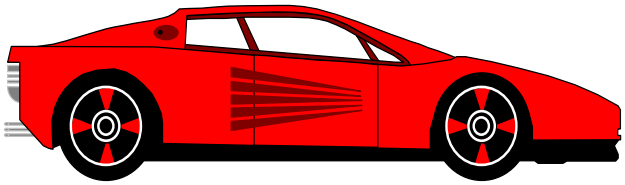
- Example: low detail but high intersection

design



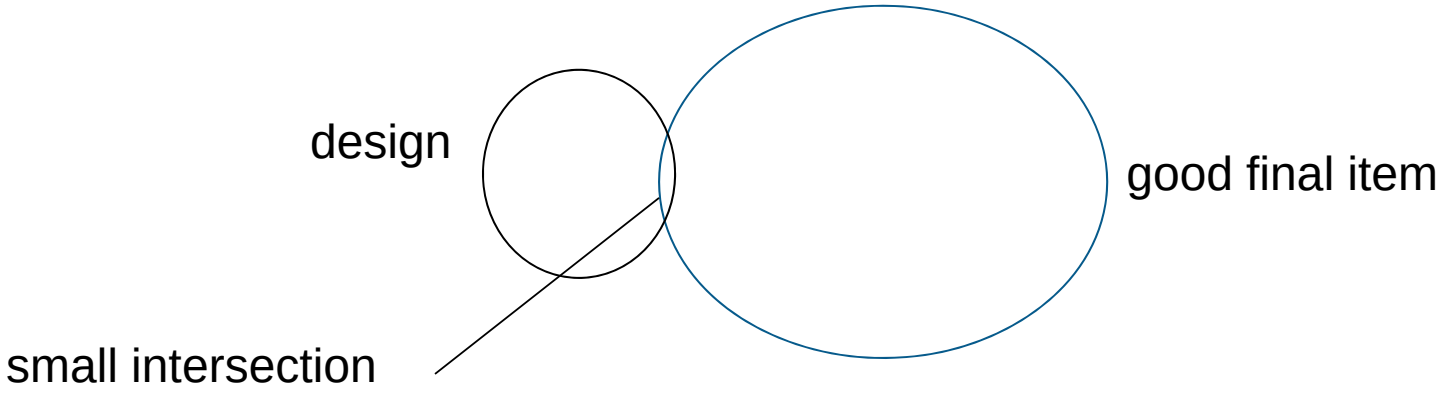
good final item





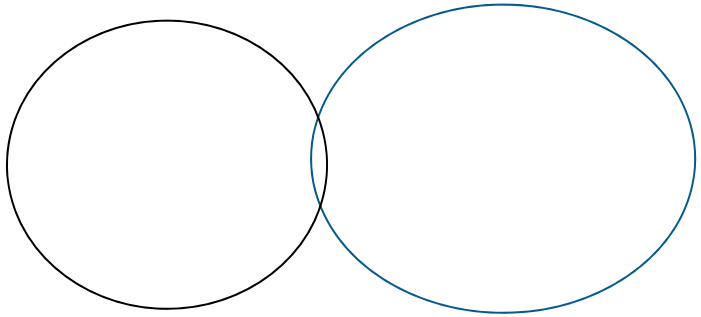
Low Intersection

- Low intersection designs
  - are somewhere down a bad branch in the design tree
  - are unlikely to satisfy the basic functional design goals
  - probably won't 'work'
  - may lead to a disastrous project

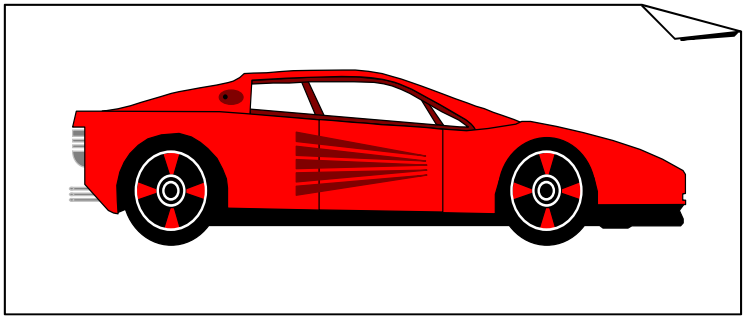


Low Intersection

- Example: high detail but low intersection
  - basic functional design goal was to provide transport over water



design



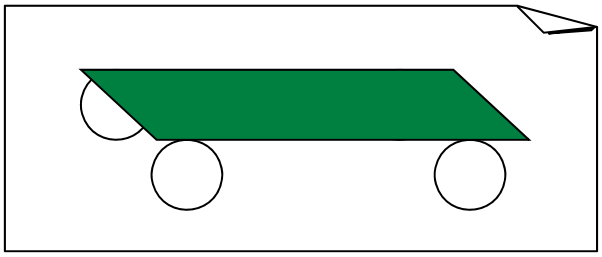
good final item

3. Merit

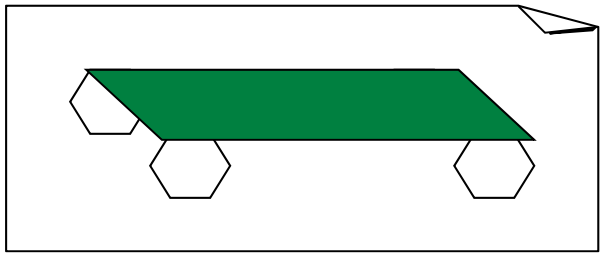
- How many desirable properties does the design have?
  - in the context of the design goals
- Examples of desirable properties:
  - might want it to be efficient, cheap, cunning, elegant, fast, light, effective, easy-to-use, attractive, bright, reliable, maintainable, powerful, strong, pliable, adaptable etc. etc.

High Merit

- Suppose transporting groceries is a design goal
  - either round or hexagonal wheels will do the job
  - but round wheels are smoother
  - smoothness is a desirable property (stops your eggs breaking)



high merit design



low merit design

High and Low Merit

- High merit designs
  - often have something intuitively appealing about them
  - e.g. enjoy the simple yet elegant effectiveness of an egg shell!
  - high merit often goes hand-in-hand with cunning simplification
- Low merit designs
  - may be cumbersome, inefficient, ugly, unreliable etc.
  - are unlikely to satisfy the performance design goals
  - won't win you any prizes or friends

Design Criteria Summary

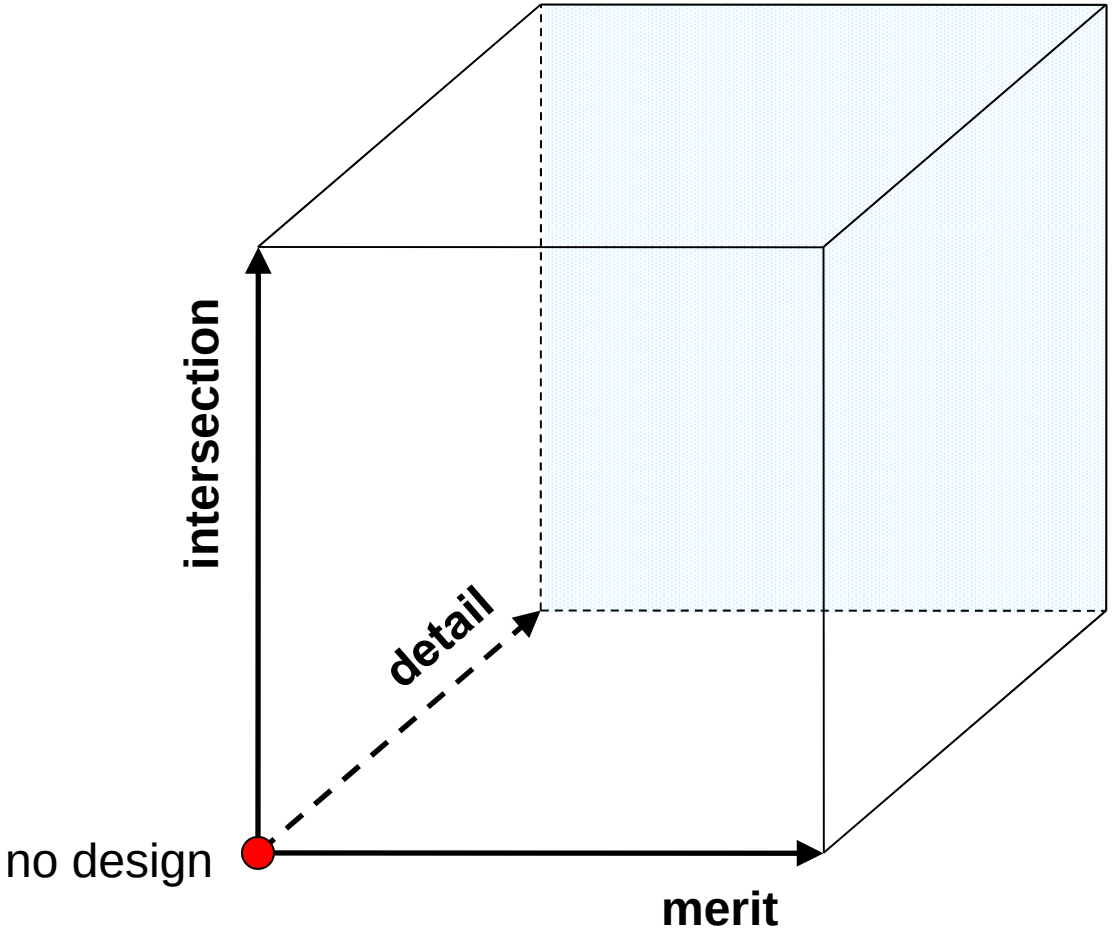
- Detail
  - without it you can't make any predictions about satisfying the design goals
- Intersection
  - without it you won't satisfy functional design goals
- Merit
  - without it you probably won't satisfy performance goals

- If you're confident you have all three
  - then you probably have a good design! – not guaranteed

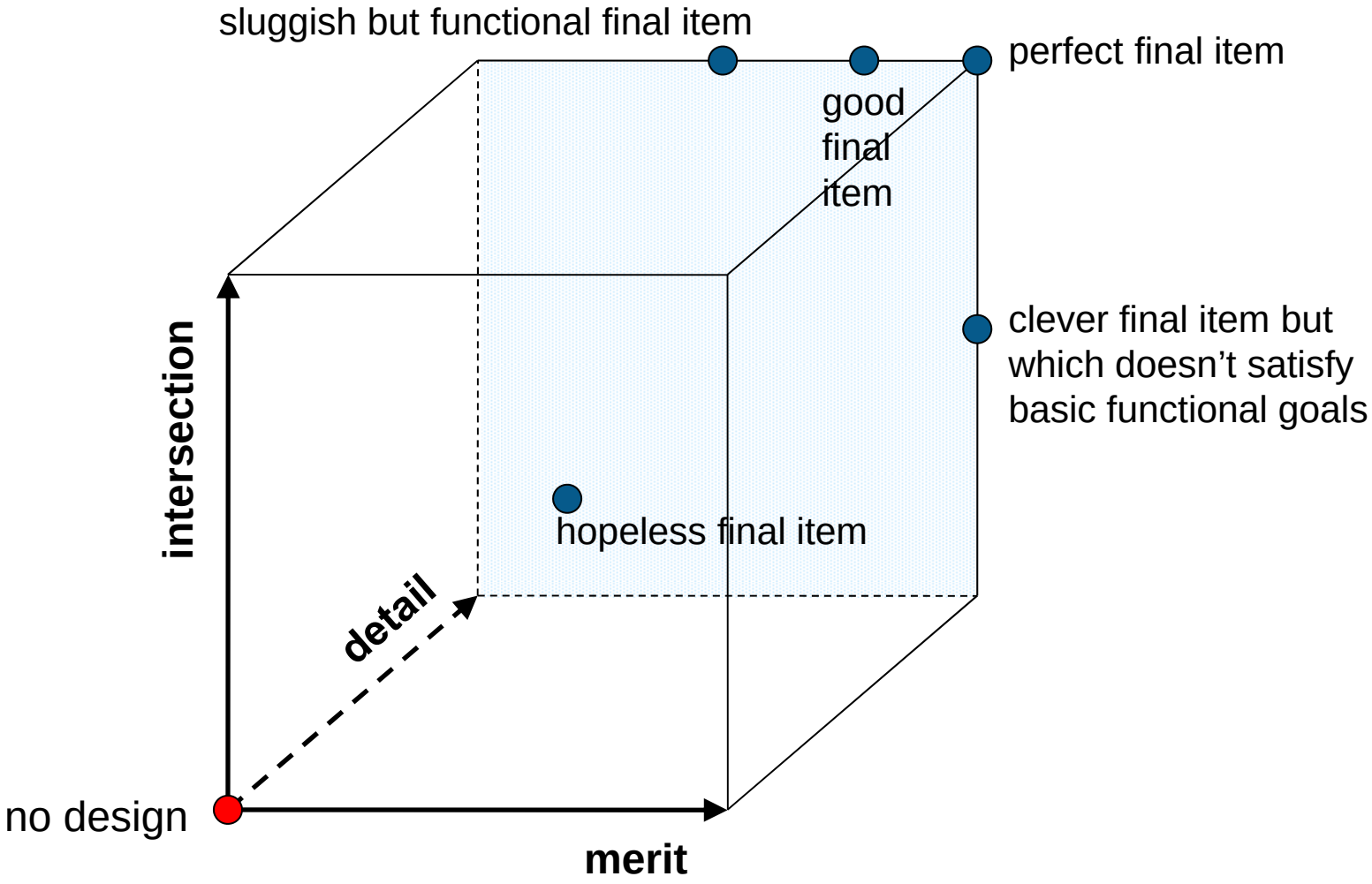
The Design Cube

- Visualisation aid for design quality
  - based on the Big 3 Design Criteria
  - tilt the design tree on its side
  - add axes for detail, intersection and merit to make a cube
  - consider where your final item and designs are inside the cube

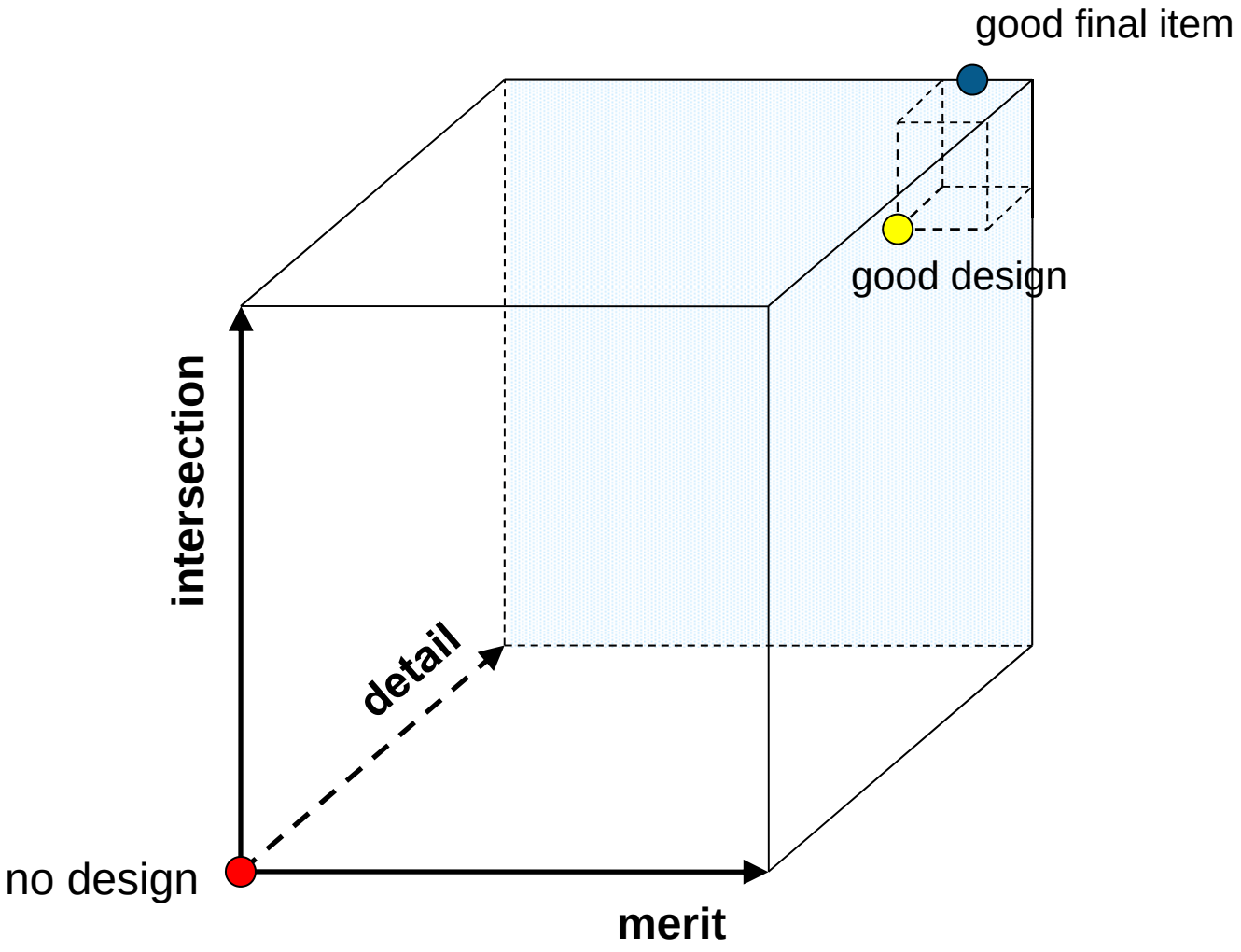
The Design Cube



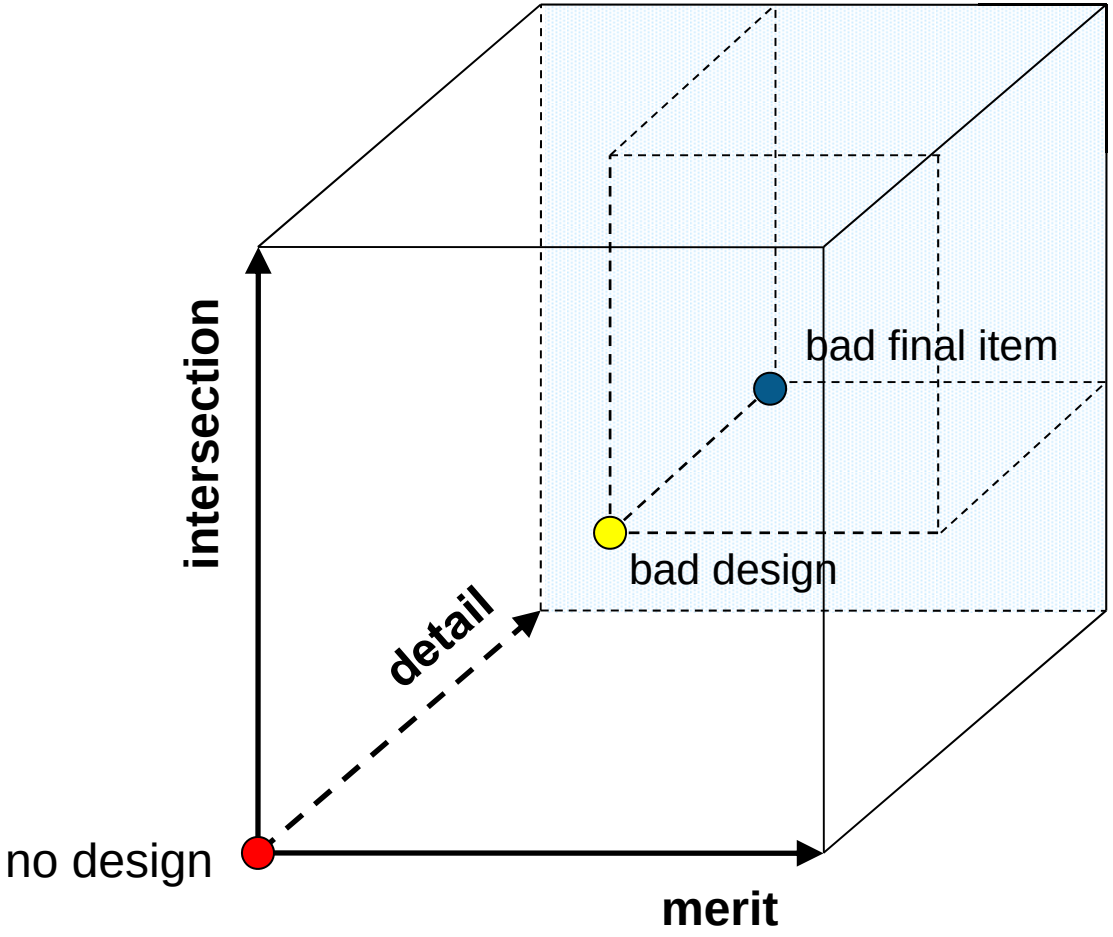
Final Items


sluggish but functional final item
perfect final item
good final item
clever final item but which doesn't satisfy basic functional goals
hopeless final item
intersection
detail
no design
merit

Good Case


good final item
good design
intersection
detail
no design
merit

Bad Case


intersection
bad final item
bad design
detail
no design
merit

Design decomposition

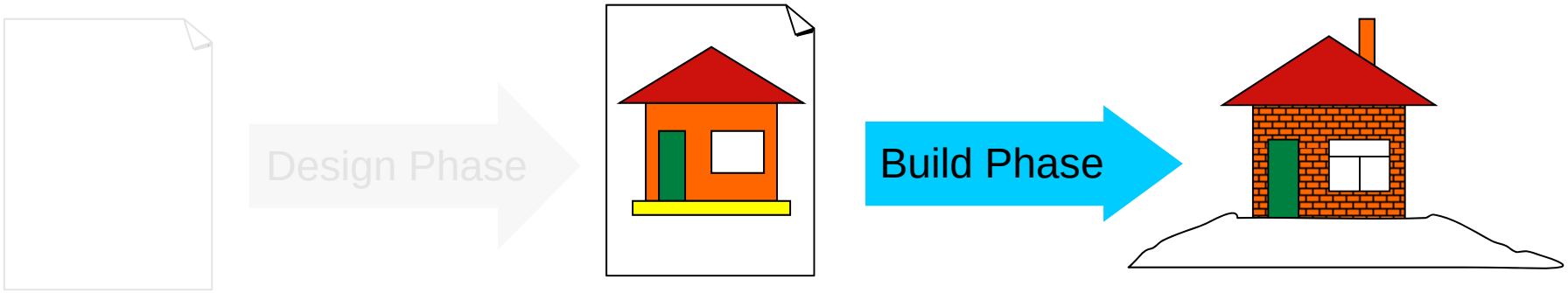
- As level of detail increases back-tracking becomes more expensive. (You have invested more effort in the work you are throwing away).
- Early in the design process try to decompose the design into component designs.
  - Minimise dependencies between components.
  - Pay special attention to designing component interactions.
- Hopefully required design changes will be restricted to a small number of related components.

Design Phase Summary

- The design phase is an iterative refinement process
- Design evaluation is a key skill
  - need it to refine a design into a better one
- But it's difficult
- Get some insights into design quality from
  - the Big 3 Design Criteria of detail, intersection and merit
  - the design cube

The Build Phase

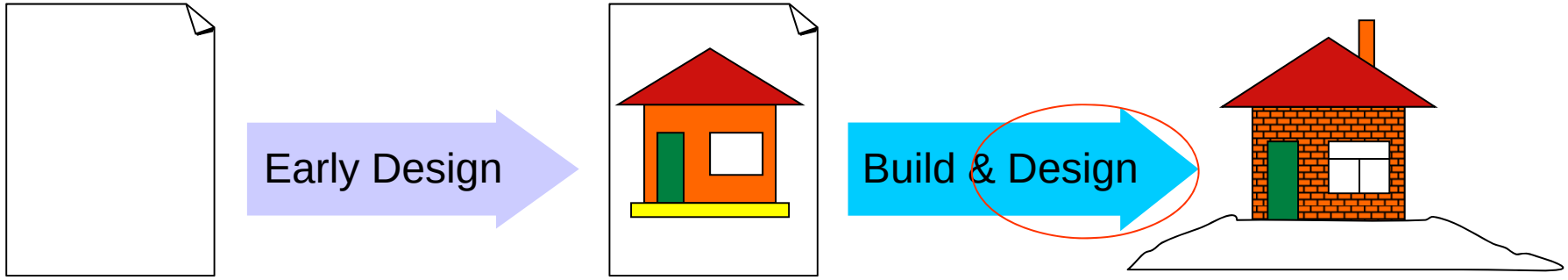
- Using the design to build the item



The Build Phase

- But some detail will be missing from the design
  - the design is an approximation after all
- And there may be defects in the design which aren't discovered until you start to build
- Have to identify and correct those defects as you go
- So what are you doing?
- Designing!
- The design continues to evolve during building

A Project Revisited

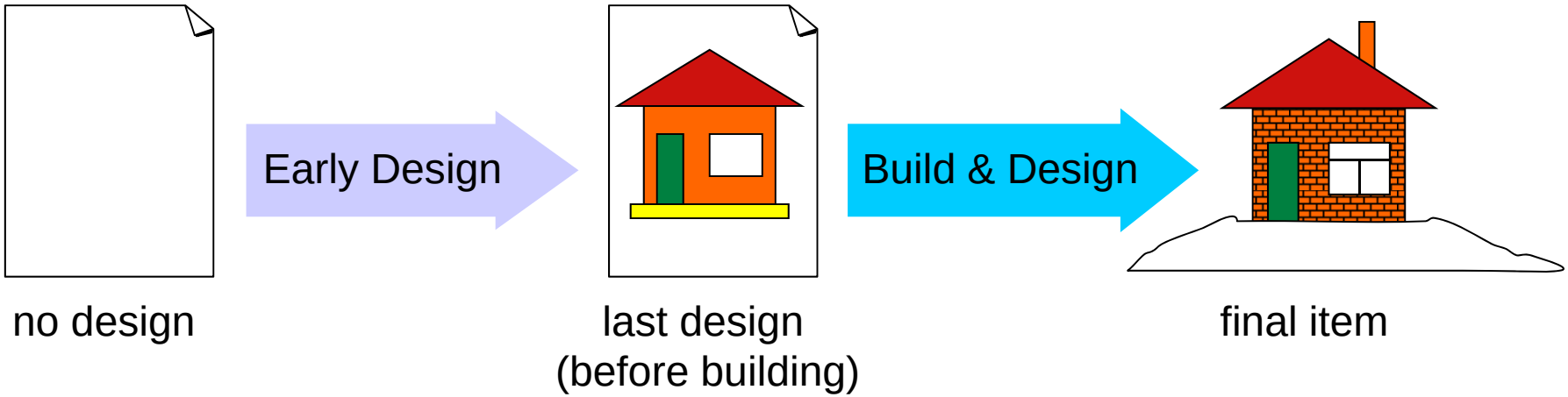


- If design is difficult, creative and uncertain, then the whole project must be too!
  - something for the project manager to bear in mind
  - one reason to become really good at design

Designing and Building

- The distinction between designing and building may be blurred further still
- Prototypes may be used to test early designs
  - probing to the next level of detail gives confidence of intersection
  - or if it doesn't work, at least you'll know there is no intersection!
- Design changes made during building may (should!) be fed back into the documented design
  - to keep the project history accurate
  - to make it easier to extend or enhance the design later

A Project Revisited



Software Design

- What is software design?
- Traditionally it's is a single phase of activity near the start of a software project – modern software engineering does not take this view

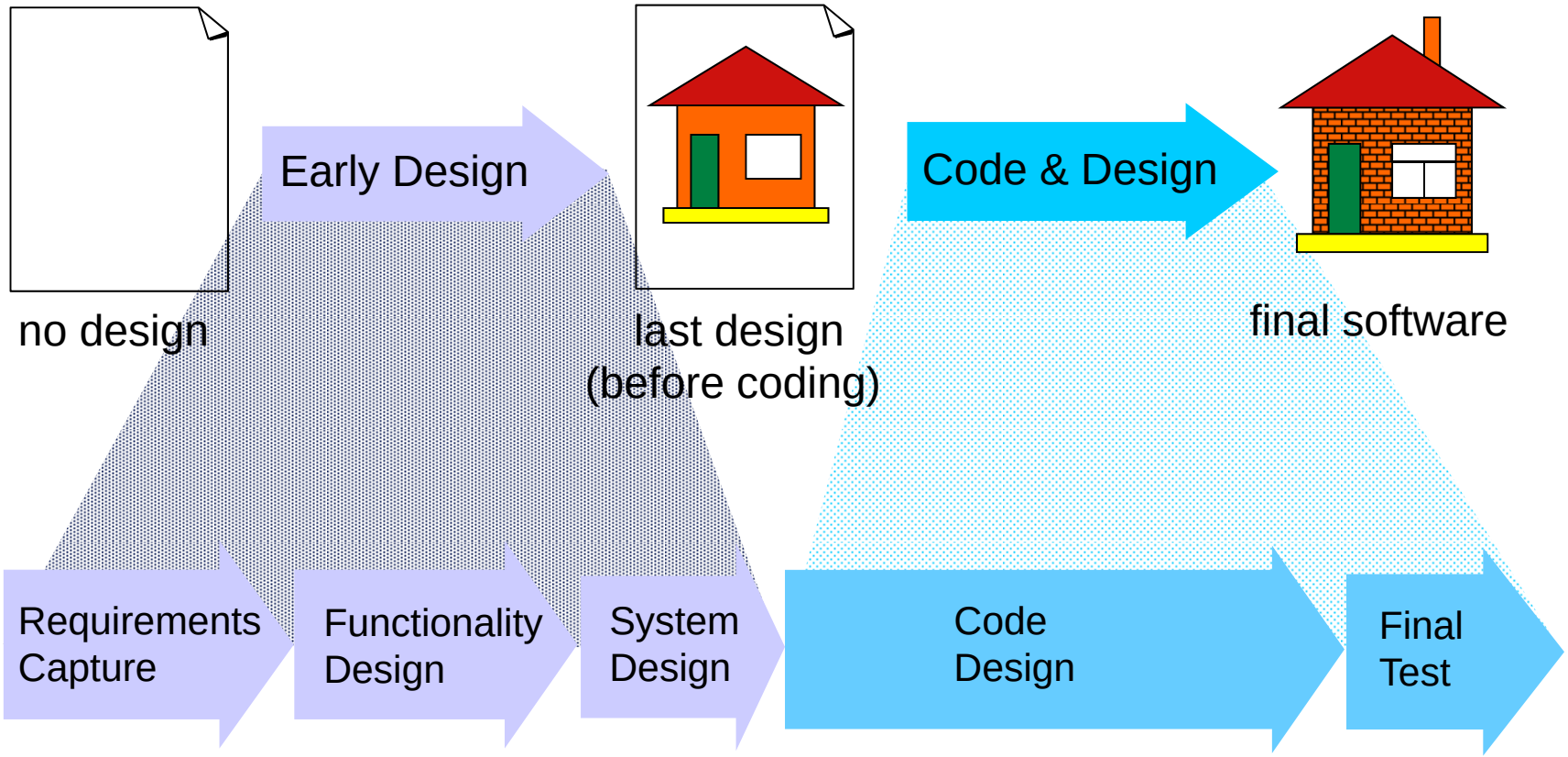
Traditional Phase	Activity	Type of Design
"Requirements"	Deciding what the goals are and what functionality to provide	Functionality Design
"Software Design"	Decomposing the system into components	System Design
"Implementation"	Writing a class (in OO), function or subroutine	Code Design
"Test"	Constructing a test to exercise a particular feature	Test Design

Software Design

- As in general projects, design work is involved *throughout* software projects
- Every decision about the software from *what it does* to *how it does it* is a software design decision
  - functionality, user interface, system architecture
  - choice of programming language, data formats/structures
  - module/class/object decomposition
  - variable naming etc. etc. etc.
- It's an ongoing activity
  - starts when the product is conceived
  - finishes the day the software gets its final release

Software Design Phases

- Replacing ‘building’ with ‘coding’:

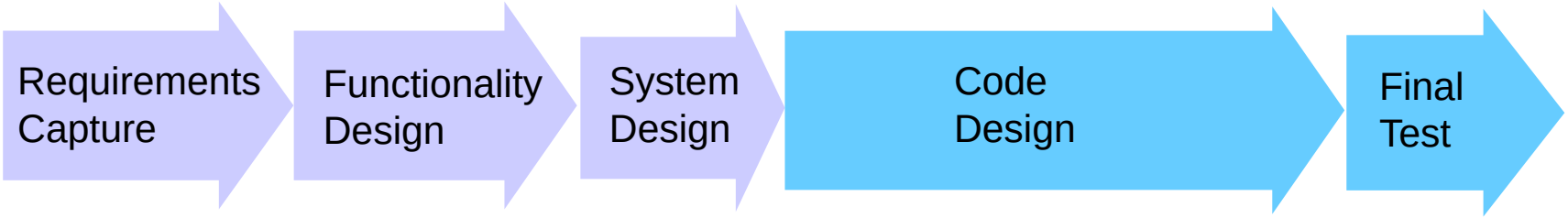


Software Design Phases

- Requirements capture
  - analyse the problem being addressed and establish the design goals
- Functionality design
  - design the behaviour of a system which will satisfy the design goals
- System design
  - design the architecture of the system and its components
- Code design
  - detailed design of the lines of code that make up the components
- Final test
  - testing (in some form) applies to all the previous phases
  - this is a final test phase before software release

Software Design Phases

- Not necessarily a rigid ordering of phases
- Software development models
  - Waterfall, Staged Delivery, Evolutionary Prototyping etc.
  - might backtrack to a previous phase on discovery of major flaw
  - might do a bit of each then repeat the cycle



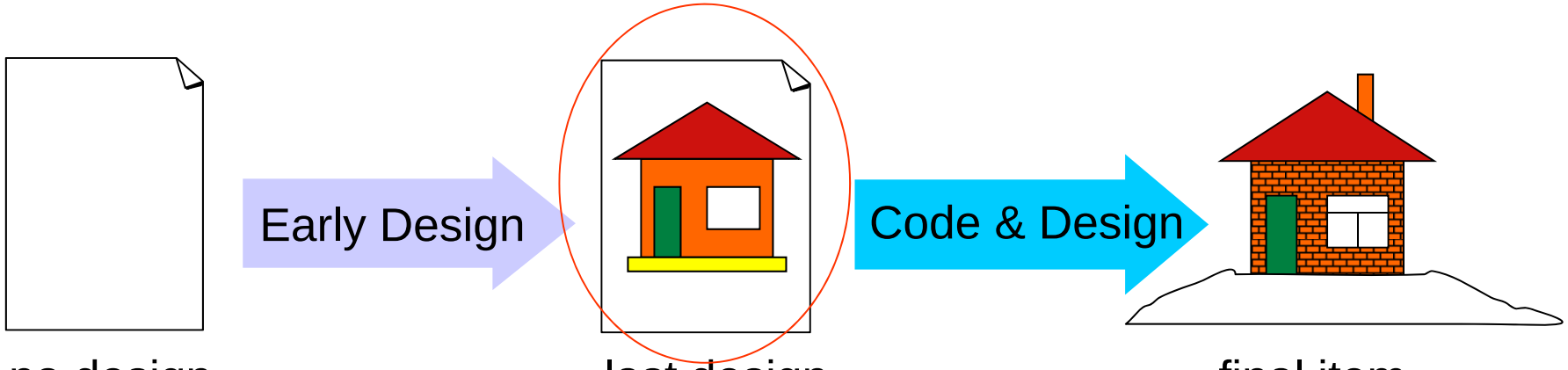
- But these are the main activities and this is the natural ordering

Early Design Overview

- Early design
  - what you do before you start coding and why it's important
- Three main activities
  - requirements capture
  - functionality design
  - system design
- Describing designs
  - tricky problem
  - text, pictures, formal diagram techniques e.g. UML
- Conclusions

Software Design Phases Revisited

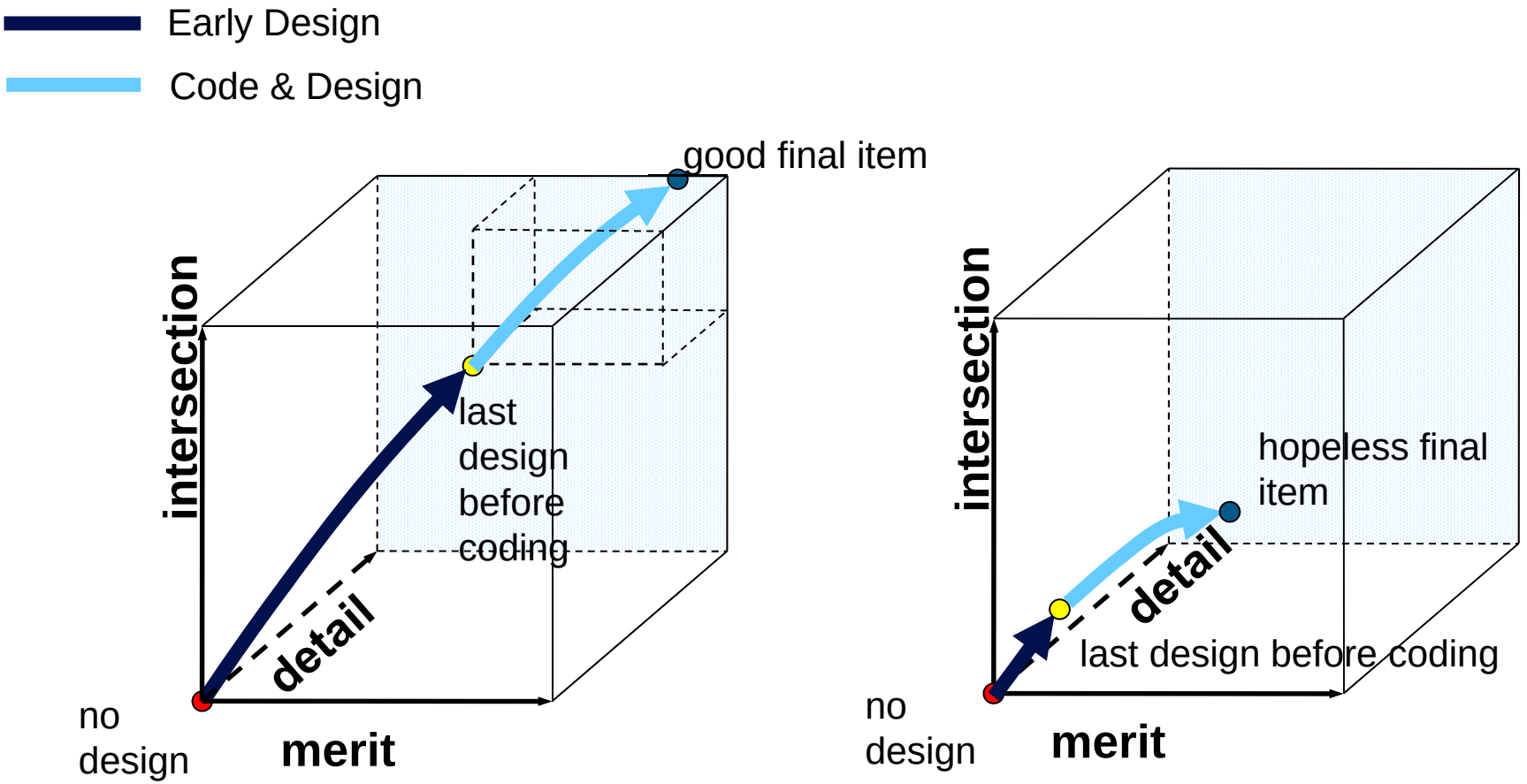
- Want to make sure that the ‘last design before coding’ is a good one



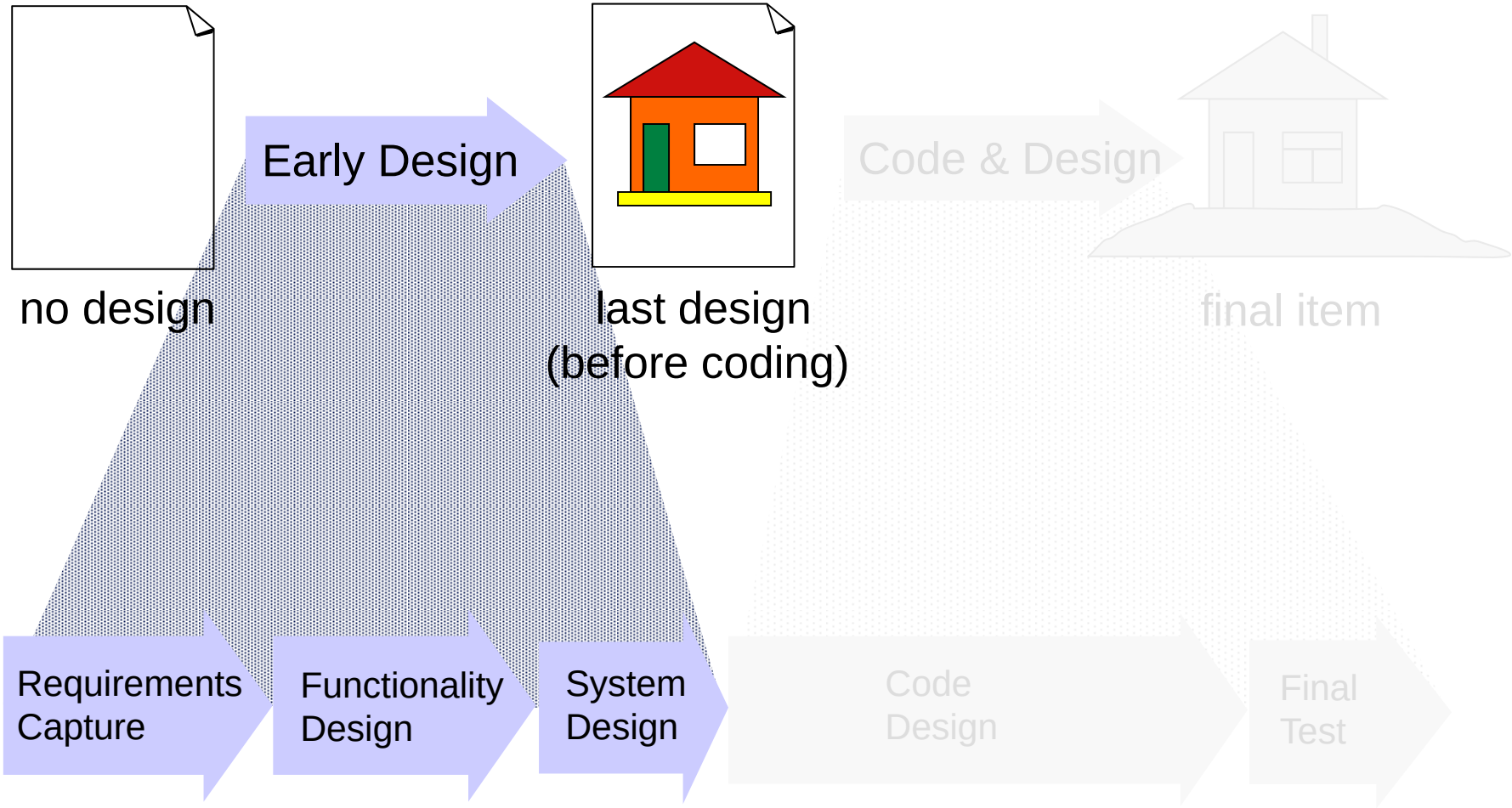
Early Design

- What happens if the last design before coding is poor or non-existent?
  - you'll be coding without a clear idea of what you're trying to achieve and why and how
  - you'll be moving to fiddly detail before getting the basics sorted
    - You need to walk before you run
  - you'll hit problems continuously, and fixing them will be costly
    - The later a change is, the more expensive
  - everything will take longer and the outcome will be poorer
  - A big reason for failure and overspending

The Design Cube Revisited



Software Design Phases Revisited



Early Design

- 1. Requirements Capture
  - "what exactly is the problem we're trying to solve?"
  - analyse the problem and establish the design goals
  - results in a requirements document
- 2. Functionality Design
  - "what's the solution going to do?"
  - functionality and user interface
  - results in a functional specification document
- 3. System Design
  - "how's it going to do it?"
  - system architecture and detailed design to some level
  - results in a system design document

Document Production

- The outcome of each of these phases is a document
- The length of these documents should be proportional to the size of the project
- But every project should have them in some form
  - the benefits of knowing what you're trying to achieve beforehand cannot be overestimated!

1. Requirements Capture

- “What exactly is the problem we’re trying to solve?”
- Aim to produce a Requirements document including:
  - Problem Statement
  - Functional Goals
    - Basic
    - Secondary
    - Enhancements
  - Performance Goals
  - Non-functional Requirements

Obtaining Information

- Who is your ‘customer’?
  - external organisation
  - internal department
  - funding body
  - research colleague
  - focus on whoever gives the ‘thumbs-up’ at the end

- Use all appropriate means to probe for accurate detailed information about the problem
  - face to face discussions
  - observation of existing system (if any)
  - study of existing documentation (if any)
  - questionnaires

The Unreliable Narrator

- Customers are like unreliable narrators in novels
  - you may get a mixture of truths, half-truths and outright falsehoods!
  - you may get conflicting information
    - particularly when several people have a say
  - information may be withheld (inadvertently or otherwise)
- But if the software solves the wrong problem, the customer will blame *you*!
- So try to untangle the requirements mess as early as possible
  - probe into the dark corners
  - overturn the stones

Cans of Worms

- Retailer: “I want a simple program to print out reports of all my current stock”
  - what's the input data?
  - how's it going to be entered? manually? bar-code swiping?
  - how is the stock data to be stored?
  - what sort of reports do you want? sorted? grouped?
  - how often do you want them generated?
  - what if it takes 5 minutes to generate? is that too long?
  - do you really mean print to a printer or to the screen?
  - what if there are reams and reams of it?

The Underlying Problem

- The customer's perception of the problem may not reflect the real underlying problem!
  - what the retailer really wanted to know was "Do I have a TX354-2 out the back?"
  - he was going to manually scan through the list of stock until he came to TX354-2 in the part number column
  - the underlying problem was the ability to query a stock database
- Need to understand the underlying business or technical problem that needs to be solved

Requirements Summary

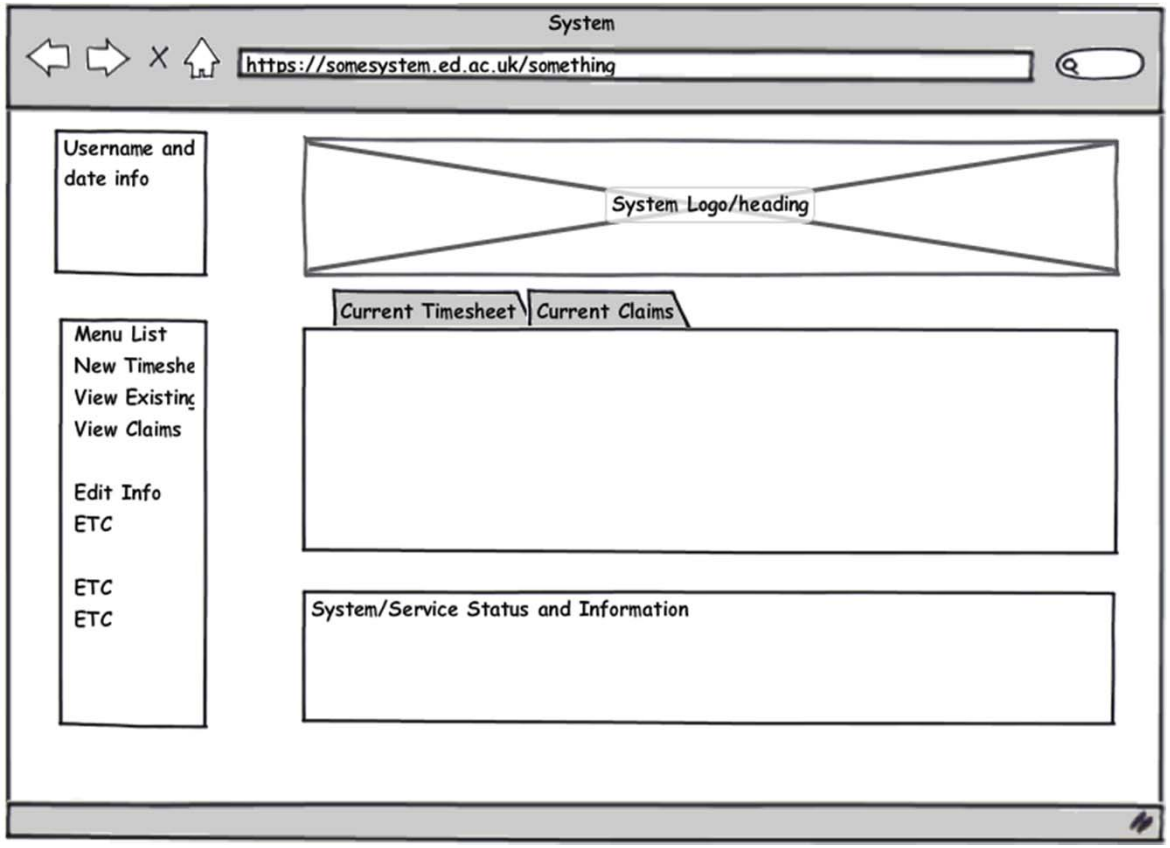
- Gather the information you need
- Resolve conflicts and inconsistencies
- Write a clear and concise Requirements document
- Seek the customer's approval of the document before proceeding

2. Functionality Design

- "What's the solution going to do?"
  - design the *behaviour* of a system which would satisfy the requirements
  - propose a software solution without worrying unduly (yet) about how to build it
- Aim to produce a Functional Specification document including:
  - the main features of the user interface
    - and how the user will interact with the UI to achieve their tasks (use model)
  - the input data
    - and how the system will modify it
  - the main functionality
    - and how it will operate on the data in order to satisfy the requirements

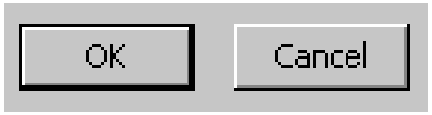
User Interface Design

- Different user interfaces for different applications
- Designing the main features of the UI early on is highly recommended
- UI prototyping
  - Balsamiq
  - Lumzy
  - Pencil
  - Pen and Paper



User Interface Design

- May have to cater for different types of users
  - novice users may want to be hand-held through it
  - expert users usually want to whiz through it with as few mouse clicks as possible
- UI conventions have evolved over the years



- Save your originality for devising intuitive ways of displaying data specific to your application domain

Use Model

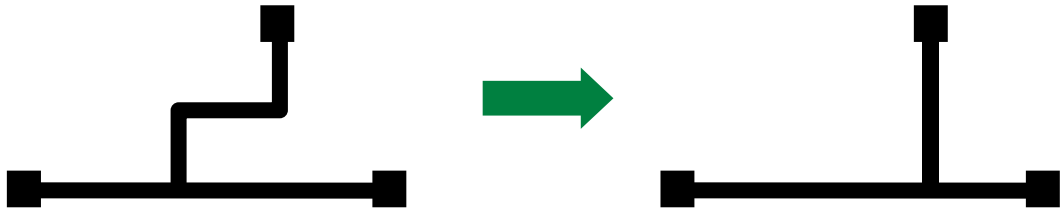
- How will the user accomplish their tasks through the user interface?
  - consider the various ‘flows’ through the software
  - document the sequences of UI interactions necessary
  - show what happens to the user’s data (files) on the way
- Can be very helpful
  - for clarifying your own ideas about how the system will behave
  - for describing to the customer how it will behave

Emphasis on Data

- In general, customers understand their data
  - it's important and precious to them
- So communicate with them in terms of things they understand
- Show them:
  - what you think their data is
  - what you're going to do to their data
  - what new data you'll leave them with at the end of the day
  - what hoops they'll have to jump through to get it
- And they'll tell you if it's a system they want

Main Functions

- What are the main functions of the system?
- For each main function describe the following:
  - its behaviour
  - its input and output data
  - how the data is modified by the function
- Use pictures and examples wherever possible
  - saves lots of typing, aids understanding
  - e.g. a “dog-leg removal” function in a chip layout program



Design Evolution Revisited

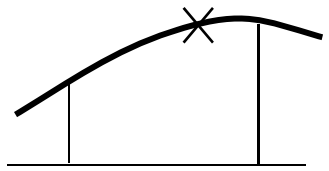
- Designing involves two main things:
  - 1. having ideas
  - 2. realising they're rubbish (and why they're rubbish)
- Iterative refinement
  - try not to fall in love with your first idea
  - through perseverance and cunning you may come up with a valuable simplification
- Encourage 'off the wall' thinking

Design Evaluation Revisited

- Detail
  - have you described the functionality in sufficient detail for it to be meaningful?
  - “and there will be a graphical user interface” is not sufficient detail!
- Intersection
  - will the functionality that you’ve described satisfy the design goals?
  - does your functionality solve the right problem?
  - is the functionality consistent and coherent?
- Merit
  - does the behaviour you’ve described have desirable properties?
  - is the system as simple as possible?
  - is it intuitive?

Design Evaluation

- Probing to the next level of detail beyond the level you're documenting can be very useful
  - helps establish the quality of what you *are* documenting
- E.g. "the interpolation facility will operate on the curve data which is passed in"
  - sounds fine
  - but on probing to the next level of detail you discover that the interpolation facility also needs a point at which to interpolate the curve
  - where's this point going to come from?
  - oops - inconsistency exposed
  - much cheaper to fix it sooner than later



Functional Design Summary

- Design the behaviour of your solution
  - and prototype the user interface if possible
- Iteratively evolve and improve the design
  - focus on the critical features first
- Try to gain confidence in its quality
  - evaluate it and get someone else to review it
- Write a clear and concise Functional Specification document describing it
- Again seek document approval from the customer

Practicalities

- Unfortunately, the functional spec could be a lot bigger than the requirements document
  - if it's more than 30 pages you may have trouble getting it read at all
  - "I've already given you the requirements, it's your job to build the system so it satisfies them"
- It's a trade-off between detail and practicalities
  - make sure that what detail you have adds value
  - emphasise the data so the customer has a direct interest
  - put the UI description and use model towards the front
    - for some reason *everyone* has an opinion on user interfaces!

3. System Design

- “How’s the solution going to work?”
  - how will the documented behaviour be realised in software?
- Aim to produce a System Design document including:
  - system architecture
    - how the system will be composed of smaller components or modules
  - component descriptions
    - responsibilities and interfaces
    - where will the main functions reside?
    - main data structures and algorithms
  - solutions to key technical problems
  - enough detail that moving to code doesn’t seem like a huge step!

System Architecture

- ‘Top-down’ approach is common
  - split the system into components or modules
    - user interface
    - database
    - core functionality
    - options package etc.
  - split the components into sub-components and so on
- Identify the interdependencies between the different components
  - try to group data and functionality so as to keep dependencies across component boundaries to a minimum

Components

- Why has this component been defined?
  - what's its purpose?
  - ensure the component has clear goals and responsibilities
- Which of the component's functions will form the interface to the outside world?
- Which of the main functions will reside in this component?

Components

- How will the component's data be modeled in software?
  - arrays, records, structs, objects?
  - what will they contain?
  - what's the lifetime of the data?
  - who's responsible for the creation / destruction of which data?
- What are the main algorithms and how will they be implemented?
  - give pseudo-code if appropriate
  - pseudo-code shouldn't just be verbose normal code!
  - Pseudo-code should help not hinder

Pseudo-Code Example

- Graphics update algorithm

```
for each open window, w {
   for each of w's objects, obj {
      if obj has been modified since last redraw then {
         redraw obj
         clear obj's modified flag
      }
   }
}
```

When to Stop?

- When should you stop documenting the system design and actually start coding?
  - tricky matter of judgement
- Things to ask yourself to see if you're ready
  - are there any parts of the design I'm particularly nervous about?
  - is my vision of the system the same as that of my co-developers?
  - is there enough design detail for coding to be an orderly guided activity?
  - do I think I'm close enough to the top-right corner of the design cube?
- Often worth going to pseudo-code detail for trickier areas first
  - quicker than writing and compiling real code

System Design Summary

- Design the architecture of the system
- Design the components and their interactions
- Evolve and improve the design
- Check it relates closely to the Functional Spec
- Write a clear and concise System Design document
- Unlike the Requirements and Functionality documents, this is an internal document
  - for the benefit of the developers when they start coding in earnest
  - customer doesn't care how it works as long as it does work

Design Issues

Keeping Documents Alive

- For some projects documents quickly ‘die’
  - they are written, looked at initially, then forgotten
- Often worth investing time updating new revisions as the project progresses
- Don’t bother adding large amounts of new detail
  - the document size will become unmanageable
  - the detail will be documented in the code after all (won’t it!)
- But try to keep them reflecting reality
  - reap rewards when someone new joins the team or someone else has to maintain the system
  - valuable history of how the requirements, functionality and system design developed over time

Describing Designs

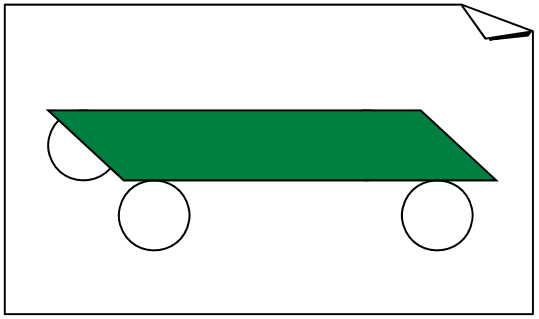
- Designs can be large and involved
  - difficult to describe concisely and consistently
  - this is a serious problem for many projects
- What are the options for describing a design?
  - text
  - pictures and diagrams
  - formal diagram techniques

Text Descriptions

- Potentially the most expressive and accurate technique
- But also the least concise
- Difficult to maintain self-consistency
- Employ the obvious ‘do’s of technical writing
  - decompose the document into sections and subsections
  - use short sentences
  - use short paragraphs
  - use bulleted lists for clarity
  - use section references to avoid duplicating information
  - read what you are writing
  - could you convey the same information in fewer lines?
  - Diagrams are your friend

Pictures and Diagrams

- People are much more willing to study pictures than text



- Quick Diagram Production
  - Balsamiq, Dia, in package tools.

Pictures and Diagrams

Joseph Jastrow, 1899 “The Minds Eye”

- Scope for confusion though
  - the reader may not interpret your picture the way you intended

Formal Diagram Techniques

- Various techniques have evolved in recent decades
  - an attempt to capture design aspects more precisely and concisely than textual descriptions
- Effectively they are visual languages
  - if everyone interprets a language in the same way then design ideas can be communicated accurately and concisely
- Well-defined semantics
  - computers can help in diagram production and consistency checking
  - some tools generate skeleton code from the diagrams
- Use them to describe designs to yourself/your team
  - don't expect customers to understand them

Formal Diagram Techniques

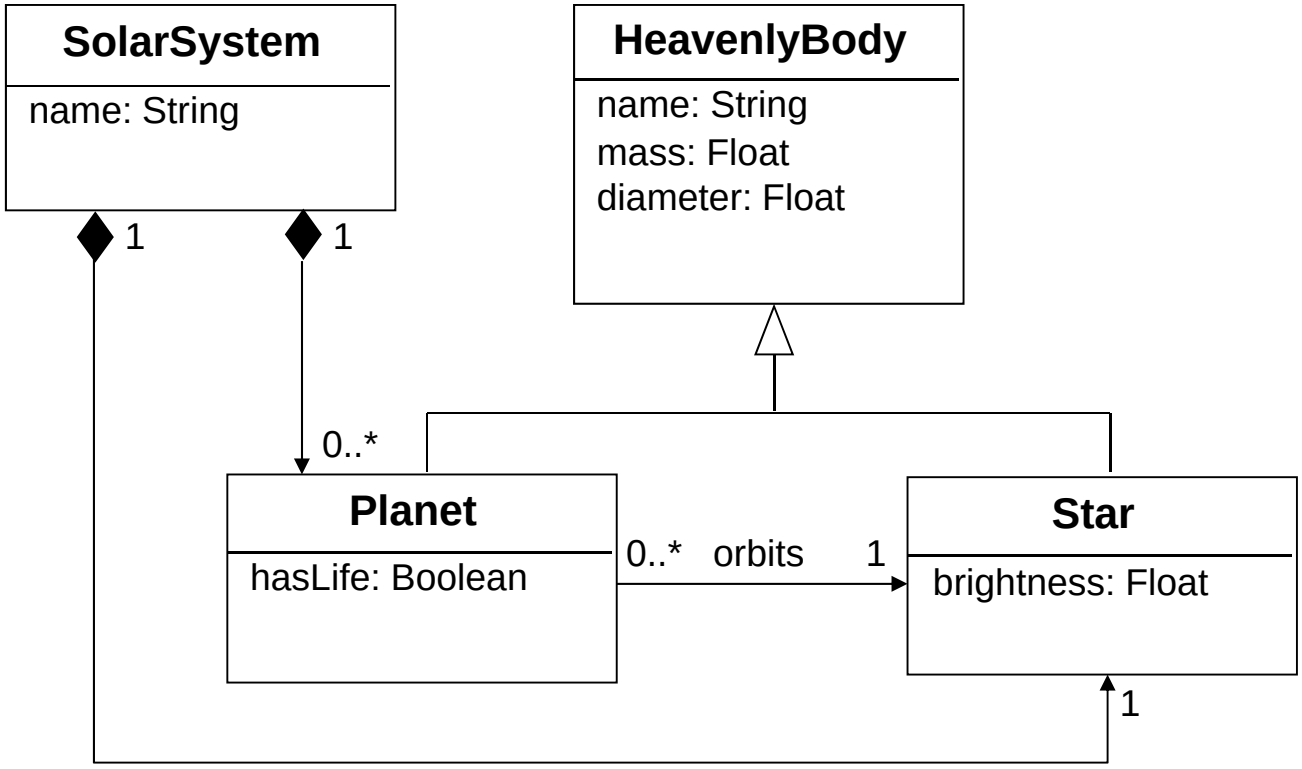
- Older ones from Structured Systems Analysis
  - Data Flow Diagrams (DFD)
    - for the transfer of data in and out of program units
  - Control Flow Diagrams (CFD)
    - for algorithms
  - Entity-Relationship Diagrams (ERD)
    - for the associations between pieces of data in a database
- Newer ones from Object-Oriented Analysis
  - Object Modeling Technique (OMT)
  - Unified Modeling Language (UML)
- UML
  - various diagram types to capture different aspects of a system

UML Diagram Types

- Class diagrams
  - shows static structure of the system's data entities (classes) and how they relate to each other
  - bit like entity-relationship diagrams
- Object diagrams (or instance diagrams)
  - example snapshot of the system's data (objects / instances) at runtime
  - "limited use, just examples"
  - *very* useful for understanding!
- Various others:
  - use case diagrams, sequence diagrams, collaboration diagrams, state diagrams etc.

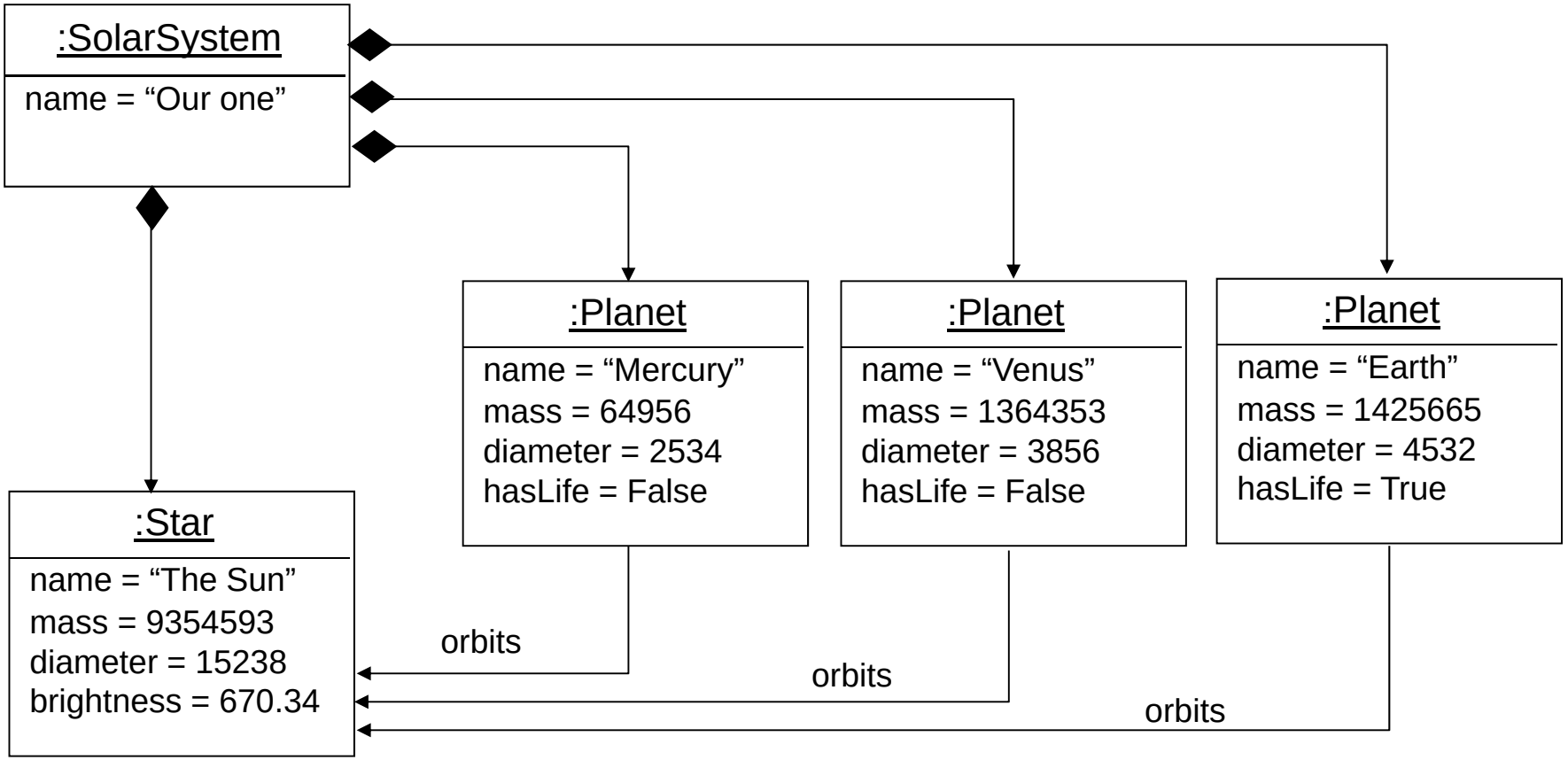
UML Class Diagram

- Solar system example



UML Object (Instance) Diagram

- Solar system example



Design Issues Overview

- Key issues for software design
  - Critical Features
  - Software Design Trade-Offs
  - Problem Anticipation
  - Cunning Simplification
  - Software Design Evolution
  - Elimination of Design Ambiguity
  - Fitting In not Forcing Change
  - Adaptation
  - Pride in Software
- Conclusions

Critical Features

- Designing non-trivial artifacts is a big problem
- Manage complexity
  - focus on essentials
- Tackle ‘critical features/requirements’ first
  - Central to the system
  - Basic functional goals
  - Highest degree of expected difficulty or uncertainty

Toaster

- Basic Function
  - To brown bread by exposure to heat
- What are the critical features?
- What are the unknowns?
- Which features are more certain?
- What are the niceties?

Pictures –

D-12 Toaster – Public Domain

Sunbeam Toaster – Donovan Govan (Creative Commons License)

Toaster

- Prioritise the feature list
  - Toasting element
  - How the bread is held

Critical Features

  - The power input
  - Safety aspects
  - Cancellation

Important Features

  - How to load the unit
  - How much is toasted at a time
  - Darkness control
  - Smoke detection

Needed Features

  - Does it defrost?
  - Aesthetics

Would be Nice Features

Toaster Critical Features

- Two critical features
- Toasting Element
  - Reusable and resilient
  - Sufficient Heat to toast the bread
- How the bread is held
  - close distance without touching
  - Repeat - should not touch the element

Text Editor Program

- Consider a simple text editor program
  - allows text insertion and deletion at the insertion point (IP)
  - can move the insertion point with cursor keys
  - allows sections of text to be given a specified font
  - save and restore
- Critical Features of the Design?
- Known/Understood Features?

Critical Features

- Data structure for the text
  - easy insertion/deletion at a current position?
  - Moving around the text in a non-linear fashion?
  - accommodate different fonts?
  - Mass font/style changes?
- Display of the text
  - How much text to display?
  - how to accommodate around the current position?
  - Re-drawing when data changes?
  - Moving around to un-displayed data?

Less Critical Features

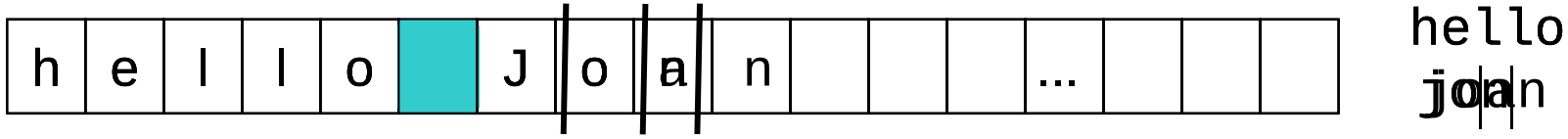
- Format for the saved file
- Deciding on the complete list of fonts
- Deciding on the kind of menu for font selection

Observations

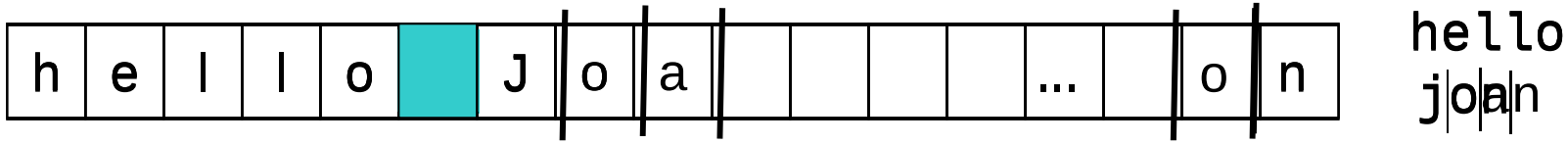
- Critical features are often
  - depended on by other features
  - Converters
  - interfaces
  - require leaps of invention
- Sometimes all the features seem critical!
  - Experience, planning, intuition help with this
- Independent features are inherently less complex and can be given lower priority

Text Editor Data Structure

- Obvious solution is a big array of characters
  - Excellent for moving the IP left or right
  - Bad for insertion and deletion

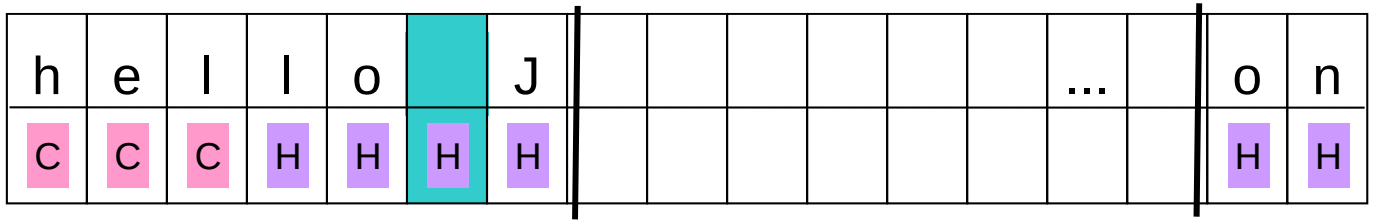


- A ‘split-stack’ approach would be better
  - Still pretty good for moving the IP left or right
  - Excellent for insertion and deletion



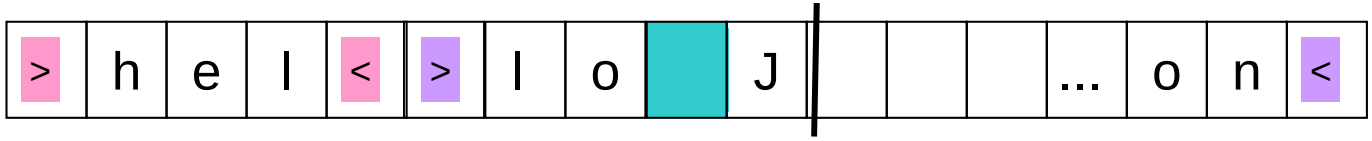
Text Editor Data Structure

- What about fonts?
  - Store the font on a per-character basis?
  - Simple solution but doubles the memory requirement



hello
jon

  - special markers for font regions?



hello
jon

  - On average reduces memory requirement
  - But what if the user changes font for ‘ello’?
  - Re-marking the regions could become tricky
  - A choice between memory and complexity

Software Design Trade-Offs

- Recall that for a design to be good we need high merit
- Common desirable software properties
  - low time (to run)
  - low space (memory)
  - low complexity (of design)
  - low difficulty (of use)

Desirable Properties

- Low time
  - users always want a faster runtime
- Low space
  - requiring too much memory will degrade the runtime (sometimes terminally) and restrict the set of suitable platforms
- Low complexity
  - simpler designs tend to be cheaper to build and easier to maintain
- Low difficulty
  - if the resulting software is not easy to use, then it is useless

Common Trade-offs

- Space vs Complexity
- Time vs Space
- Time vs Complexity
- Complexity vs Difficulty

Resolving Trade-offs

- No magic or one size fits all solution
- Evaluate relative priorities on a case-by-case basis
- In the absence of other factors, we suggest using the following facts for guidance
  - memories keep getting bigger
  - machines keep getting faster
  - life is finite and the remainders of our lives gets shorter every day

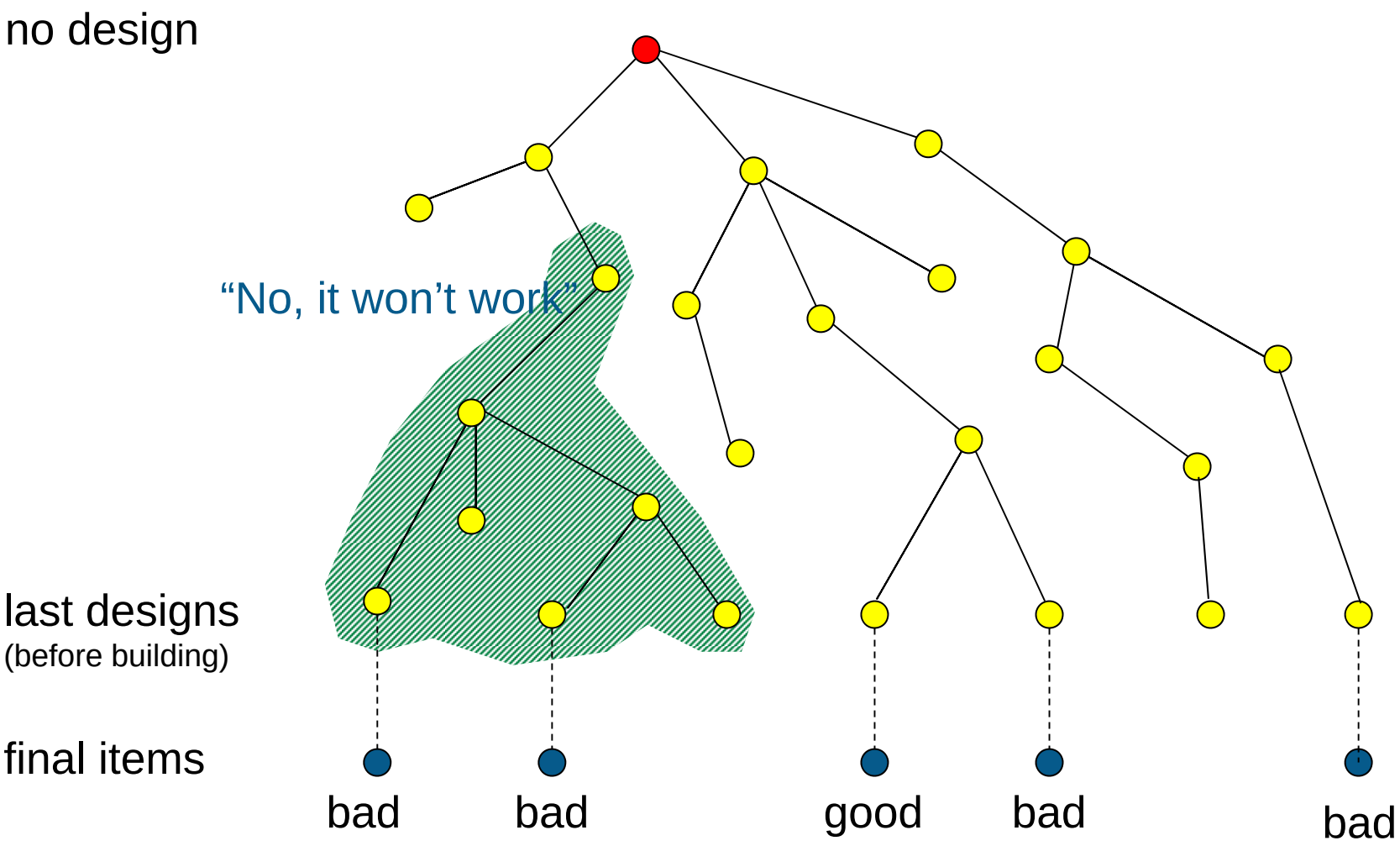
Life-Centric Priorities

- 1. Low complexity
  - high complexity complicates testing and debugging
- 2. Low difficulty
  - Difficult to use software will annoy and frustrate users
- 3. Low time
  - Software should take a reasonable time to run for the task
- 4. Low space
  - there will probably be a bigger machine next year, but do not get lazy and complacent

Problem Anticipation

- Being able to look at a proposal and say “No, it won’t work, because …”
- Careful consideration of implications of current design decisions
- Ask probing questions of the design/designer
- Can be perceived as a negative skill
- Can have very positive effects
  - wiping out whole branches of the design tree
  - improving chances of arriving at a good design
- The earlier a problem is found the better

Problem Anticipation

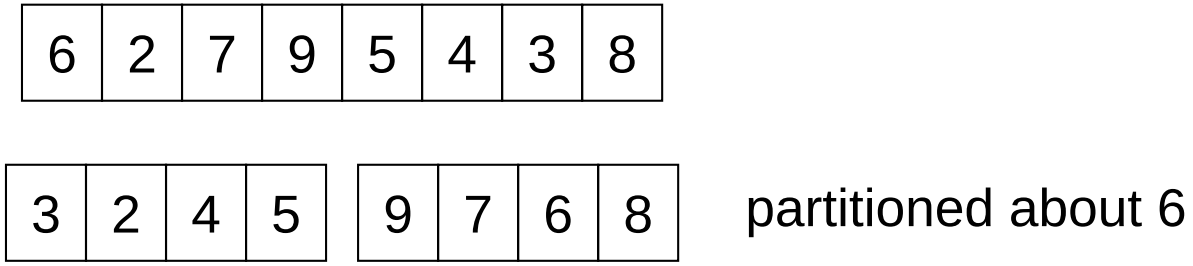


Cunning Simplification

- high merit designs are often the simplest
- Simple design does not equate to naïve design
- It is often cunning which brings about the simplicity
- Consider the classic Quicksort algorithm

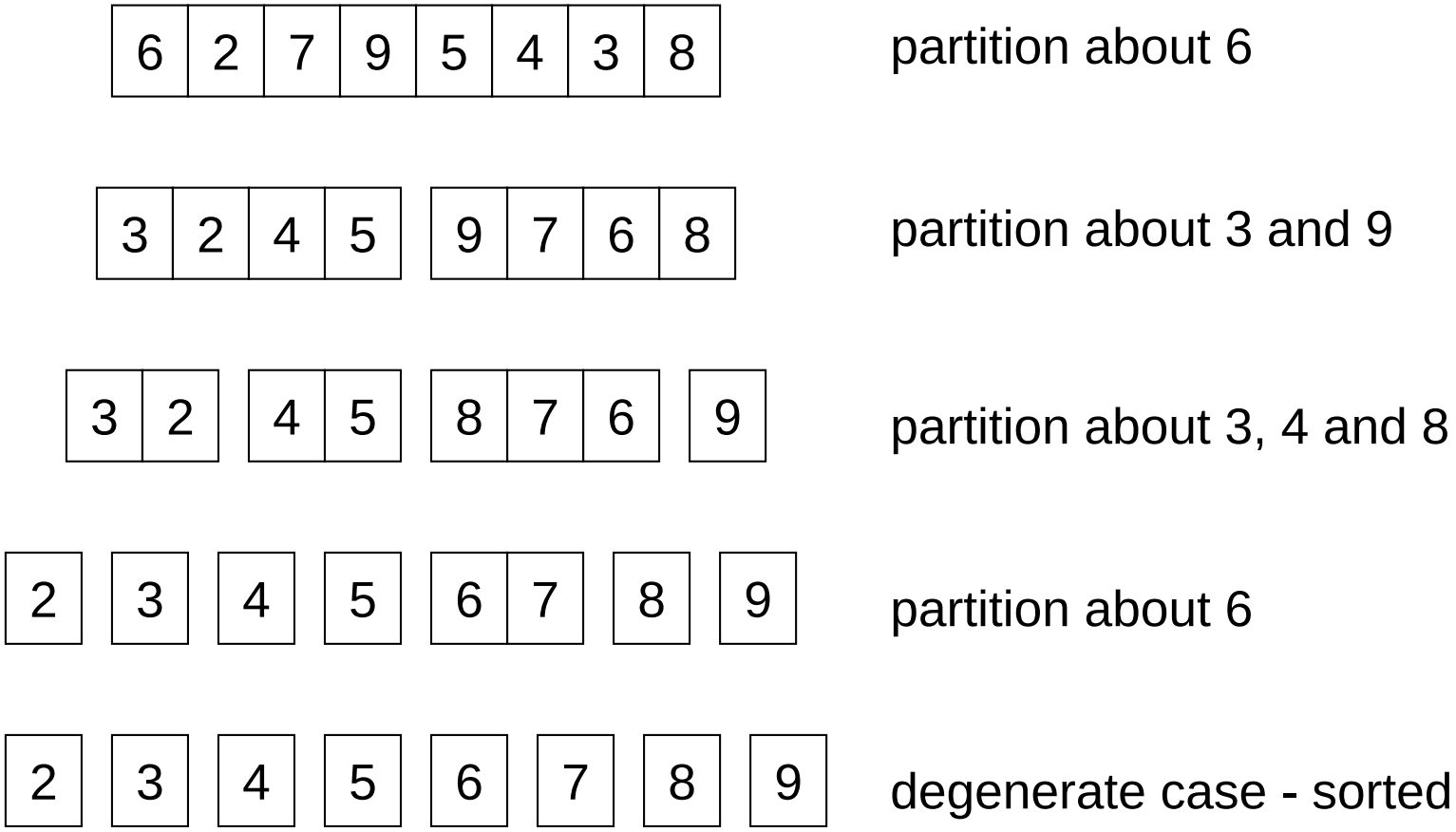
Quicksort

- Quicksort uses partitioning
  - pick a pivot element, p (e.g. the first in the array, 6)
  - start from the left and right ends and move inwards
  - swap left elements >= 6 with right elements <= 6



- Cunning observation:
  - if you partition an array, the two partitions may then be sorted *independently*
  - this simplification may be applied recursively

Quicksort Example

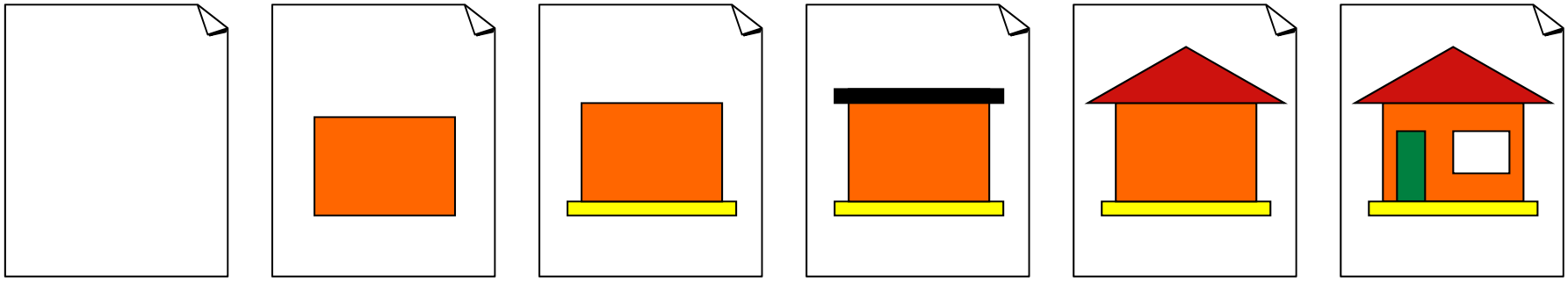


Quicksort

- The result is a very fast yet simple sorting algorithm
- Quicksort is an example of ‘Divide and Conquer’
- Detail, Thought, Ingenuity

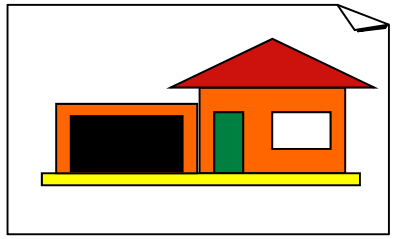
Software Design Evolution

- Designs change – more detail, change in goals …
- ‘Iterative Refinement’

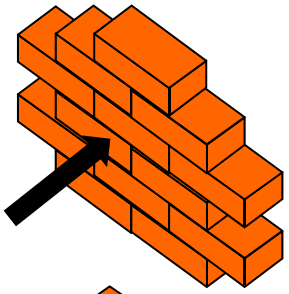


- Consistency is important

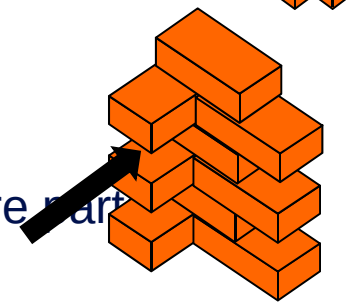
Software Design Evolution



- Next stage is to add a garage
- Bad to just abut it against the house wall



- Much better to alter part of existing wall first



- Formerly excellent pieces of design may need re-work when more part added

Consistent Designs

- Early designs tend towards
  - High intersection
  - high merit
  - low detail

- Important for Design
  - Simple
  - Elegant
  - Clean
  - Self-consistent

Consistent Designs

- When adding more detail or features aim for careful and consistent extension
- Often result is a quick hacky extension
- But what is one quick hack?
- End up with a ugly, inconsistent, unmaintainable product
- High detail but low intersection and very low merit

Getting Inconsistent

- Often appears easier to add a quick dirty ‘solution’ than do it properly
- False Economy
- Consistent evolution requires a clear understanding of the current design
- Lack of understanding breeds a reluctance to modify existing code for fear of breaking it
- Better to restructure the original code and re-test it

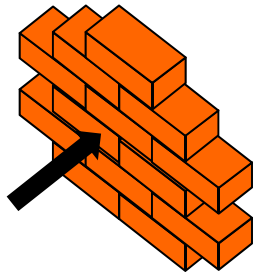
Simple Example

- Suppose there's a function to print someone's name and age, but you just want to print their name

```
void printNameAndAge (int i) {
  // Print out the name and age of the person at the argument index.
  printf("Name is %s \n", names(i));
  printf("Age is %d \n", ages(i));
}
```

- You could simply add this:

```
void printName (int i) {
  // Print out all the name of the person at the argument index.
  printf("Name is %s \n", names(i));
}
```

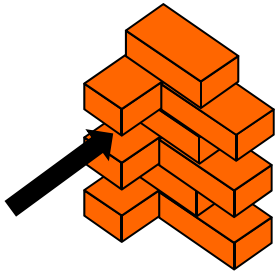


Simple Example (cont.)

- Change the original function to use your new one:

```
void printName (int i) {
  // Print out all the name of the person at the argument index.
  printf("Name is %s \n", names(i));
}

void printNameAndAge (int i) {
  // Print out the name and age of the person at the argument index.
  printName(i);
  printf("Age is %d \n", ages(i));
}
```



- Avoids code duplication
- Avoids potential for future inconsistency bugs

Trying to do it right

- What should you do when faced with extending a inconsistent code-base?
- Don't panic, and don't make it worse
- Try to get extra time to smooth out the code
  - always a good investment even though it adds no new functionality
  - 'refactoring'

Eliminate Design Ambiguity

- A component or function needs to have a clear function
- If a function can have multiple interpretations there may be problems
- Where ambiguities arise:
  - Naming
  - Responses
  - Side-effects
  - Hidden Knowledge

Naming

- Functions, Components should all have clear and unambiguous names
  - except for common methods
  - Don’t come up with multiple names of printObject or toString
- Common in database applications
- If something can be confused as to purpose or be confused with another function or variable – clarify it
- Corrections – in databases can prefix with tables and databases
- Corrections – in programming can use module prefixing or aliasing

Responses

- Try to ensure when defining responses – especially error responses that one type of response is:
  - Always consistent – HTTP 404 – requested resource not found – not server not found
    - This is a consistent across modern web platforms
  - Clearly Defined – responses should have a definite form and meaning
    - For example the result of a calculation should be a value or error
- If you cannot accurately write down what a response from a function or error handler is then something is wrong – this will lead to problems for other developers and users

Side-Effects

- Procedures can have side-effects
- Make sure these are listed and defined in the documentation and code commentary
- What are side-effects
  - Changes to program data
  - Changes to system status
  - Alteration of program behaviour
- These may not always the primary purpose of a function or procedure – it stills needs to be defined

Hidden Knowledge

- Lots of types of hidden knowledge
  - Function parameter meaning
  - dependencies
- Do not hide away function parameters
  - Default settings or values should be explicit
  - Alternative values should be explicit
- Make sure dependencies of objects and functions are explicit
- Hidden Knowledge in the development environment leads to duplication of work or erroneous use

Fitting In

- If you want people to use your software then do not force major changes in the way they work
- Software Adoption is driven by fitting in
  - Makes users feel important
  - Limits disruption of changes
  - Shows understanding of problem areas
- Major changes in work patterns will reduce user satisfaction
- Exception – where a new method of working can be proven to be better – more efficient or more accurate – then negotiate with the client

Adaptation

- Nothing stays the same
- Even in short projects the requirements can shift
- Most of the time the shift will be minor – few projects will completely change their purpose or focus
- Do not despair when a client asks for change
  - Negotiate
  - Accommodate
  - In rare circumstances Refuse
- Change is normal

Pride in Software

- Designers, Artists and Architects take pride in their work
- Great designers like to show off their work
- They learn a lot from seeing each other's work
  - Constructive Criticism of own work by peers
- Software designers can be reluctant to do this
  - Everyone makes mistakes
  - People who think they are infallible are a problem on a project

What is Design Conclusions

- Good software design will make our lives better
- We design so that items satisfy design goals
- Design is inventive, iterative, difficult and unpredictable
- Designs must be evaluated
  - to predict final item quality and to improve a current design
- But design evaluation is tricky
  - detail, intersection, merit and the design cube offer some insights
- Design is involved *throughout* a project
- There are several phases of software design

Early Design Conclusions

- Having a good design before coding gives you a much better chance of success
- Three main activities in Early Design
  - requirements capture
  - functionality design
  - system design
- Describing designs is difficult
  - use well-written concise text
  - lots of pictures and examples
  - consider a formal diagram technique such as UML

Conclusions

- Software design issues
  - identify and tackle critical features first
  - resolve design trade-offs with the project priorities in mind
  - seek out problems as early as possible
  - use cunning to simplify the design
  - keep it consistent as it evolves
  - Ambiguity
  - Fitting In
  - Adaption
  - take pride in it