zühlke
empowering ideas

Software Design

An Industrial Perspective

Lecture 7, Software Design, Part 1A CST

Mike Hogg

Who am I?

zühlke
empowering ideas

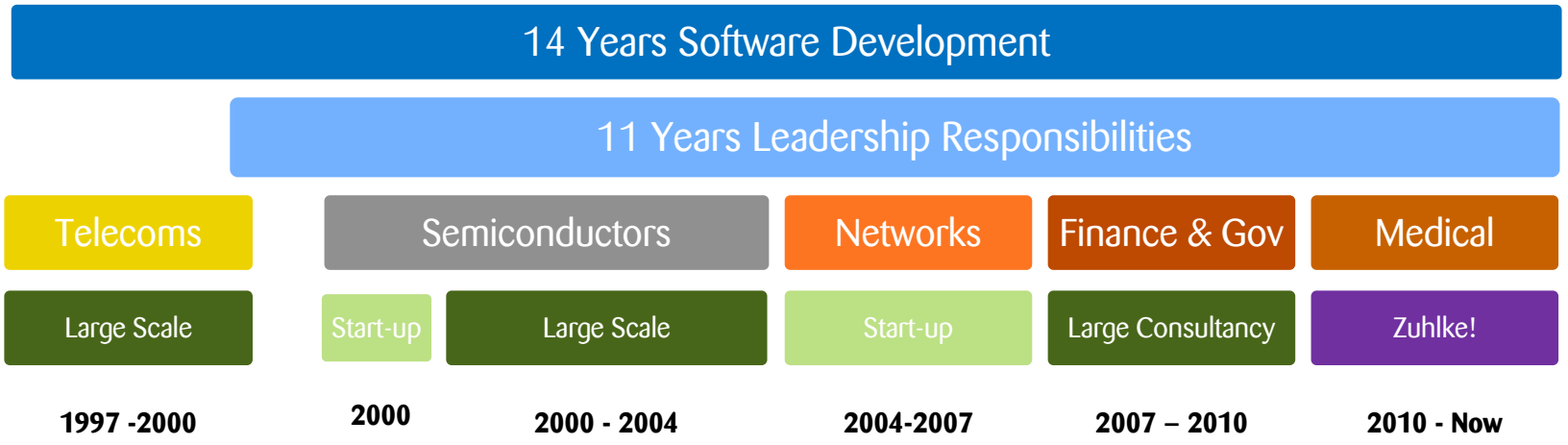
Education

- Cambridge University 1992 -1996
- MEng Electrical and Information Science

Professional Qualifications

- Chartered Engineer (CEng)
- Member of IET

Summary of Experience



Who are Zuhlke?

- Software Solutions, Product Innovation and Consulting
- Over 7000 projects delivered
- Turnover of €51M (2010)
- 400 Employees
- Active in Austria, Germany, Switzerland and UK
- Founded in 1968, owned by management team since 2000
- ISO 9001 and 13485 certified

Agenda

zühlke
empowering ideas

1. Development Lifecycles
2. Requirements
3. How much design do you need?
4. Lower level design
5. Other stuff

zühlke
empowering ideas

Development Lifecycles

Mike Hogg

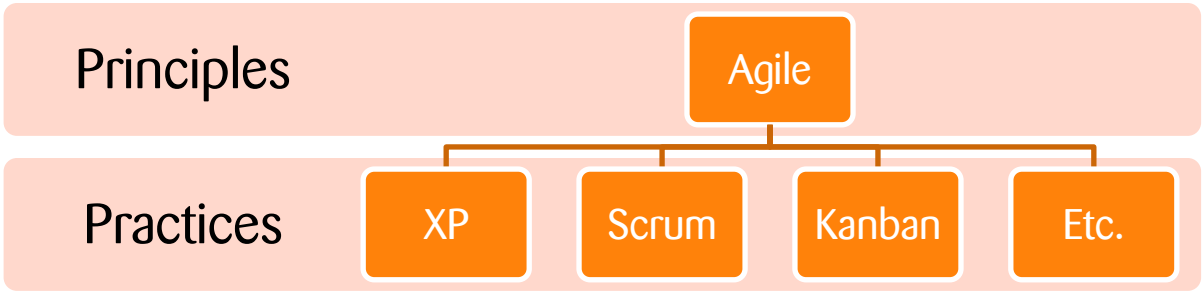
Agile Manifesto

zühlke
empowering ideas

"We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan

That is, while there is value in the items on the right, we value the items on the left more."



Development Lifecycles - Phased

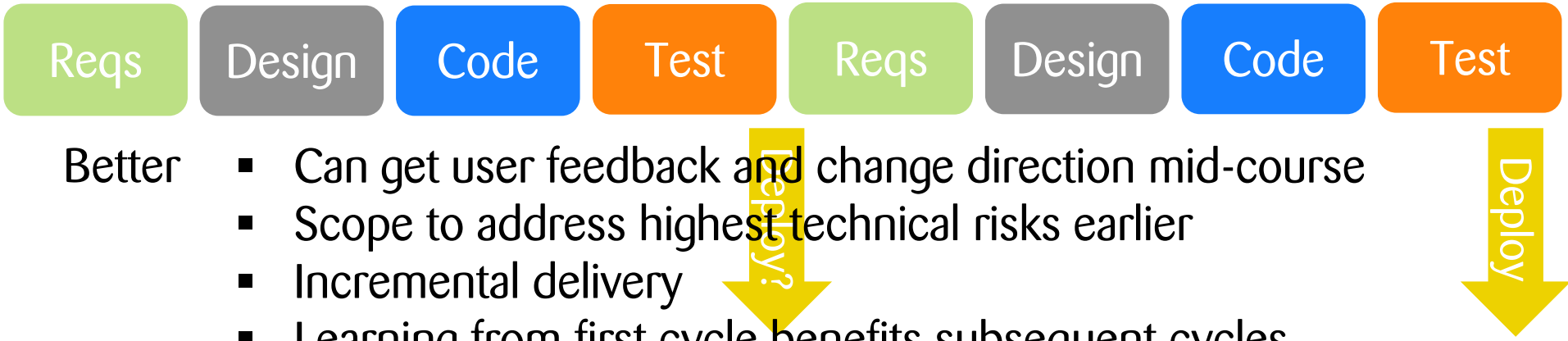
zühlke
empowering ideas

Waterfall

High risk!

- The world changes, requirements change
- Technical issues can be encountered late
- Big bang integration = painful

Large Iterations, Distinct Phases



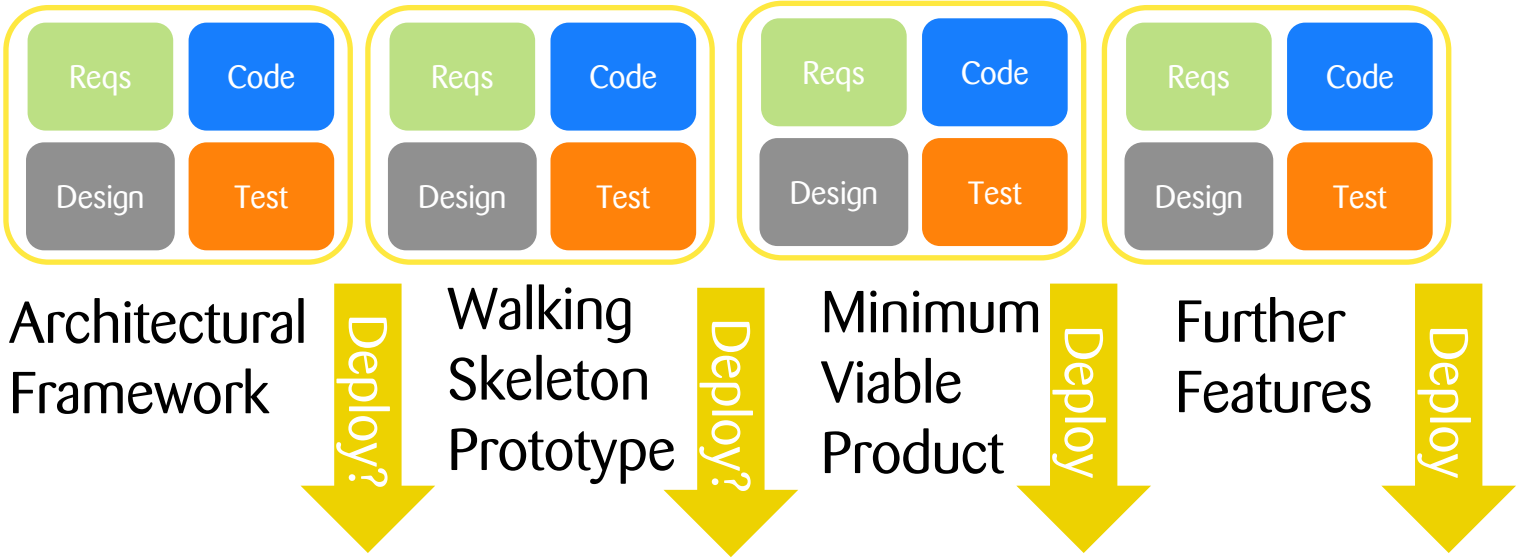
Better

- Can get user feedback and change direction mid-course
- Scope to address highest technical risks earlier
- Incremental delivery
- Learning from first cycle benefits subsequent cycles

Development Lifecycles - Agile

zühlke
empowering ideas

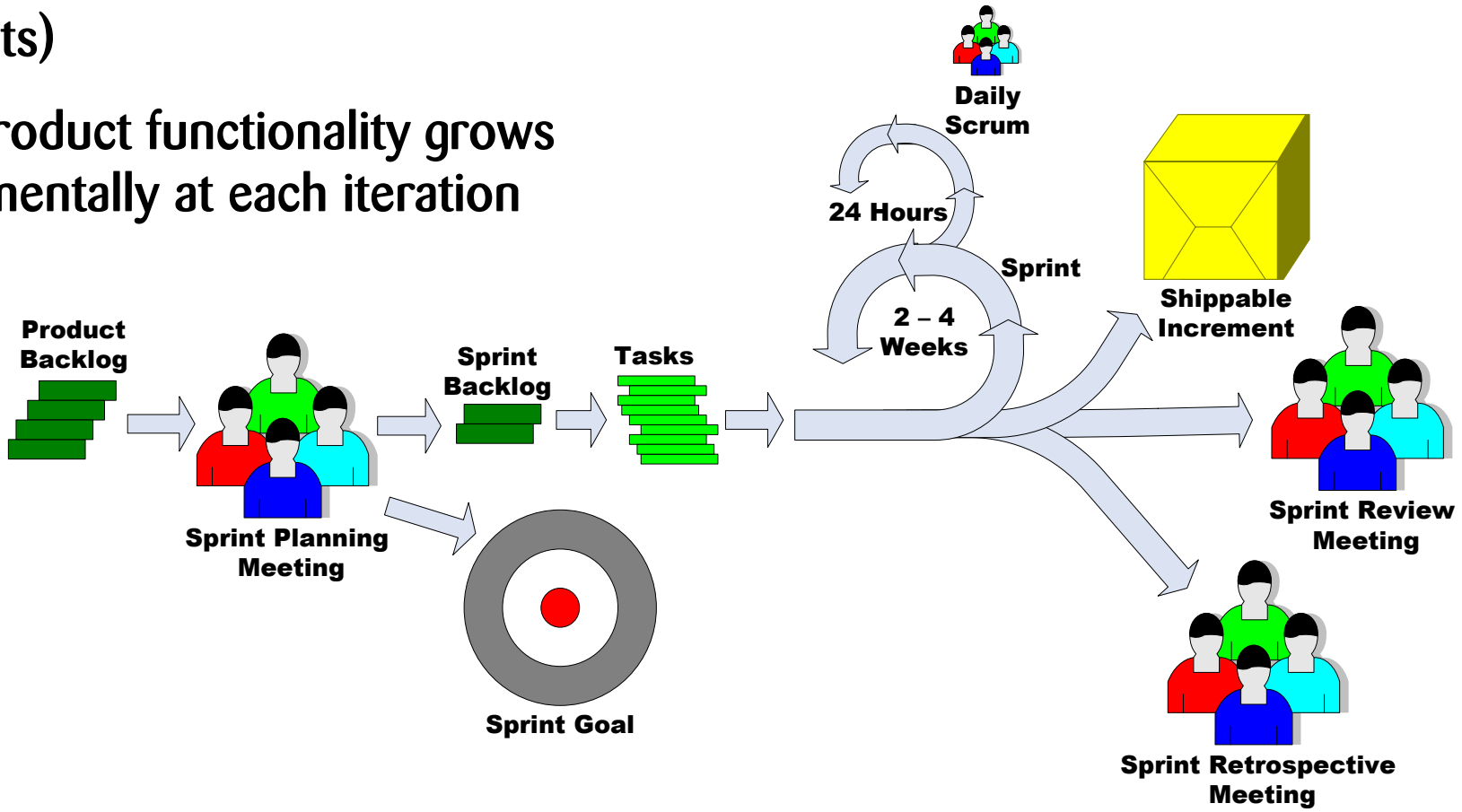
Iterative, Indistinct Phases



- Working Software:
  - Coding starts as early as feasible
- Customer Collaboration:
  - Frequent releases, on-going scope conversation
- Respond to change:
  - Scope not finalised until start of each iteration

Iterative working with Scrum

- The product is developed in a sequence of self contained time-boxes called iterations (Sprints)
- The product functionality grows incrementally at each iteration



zühlke
empowering ideas

Requirements

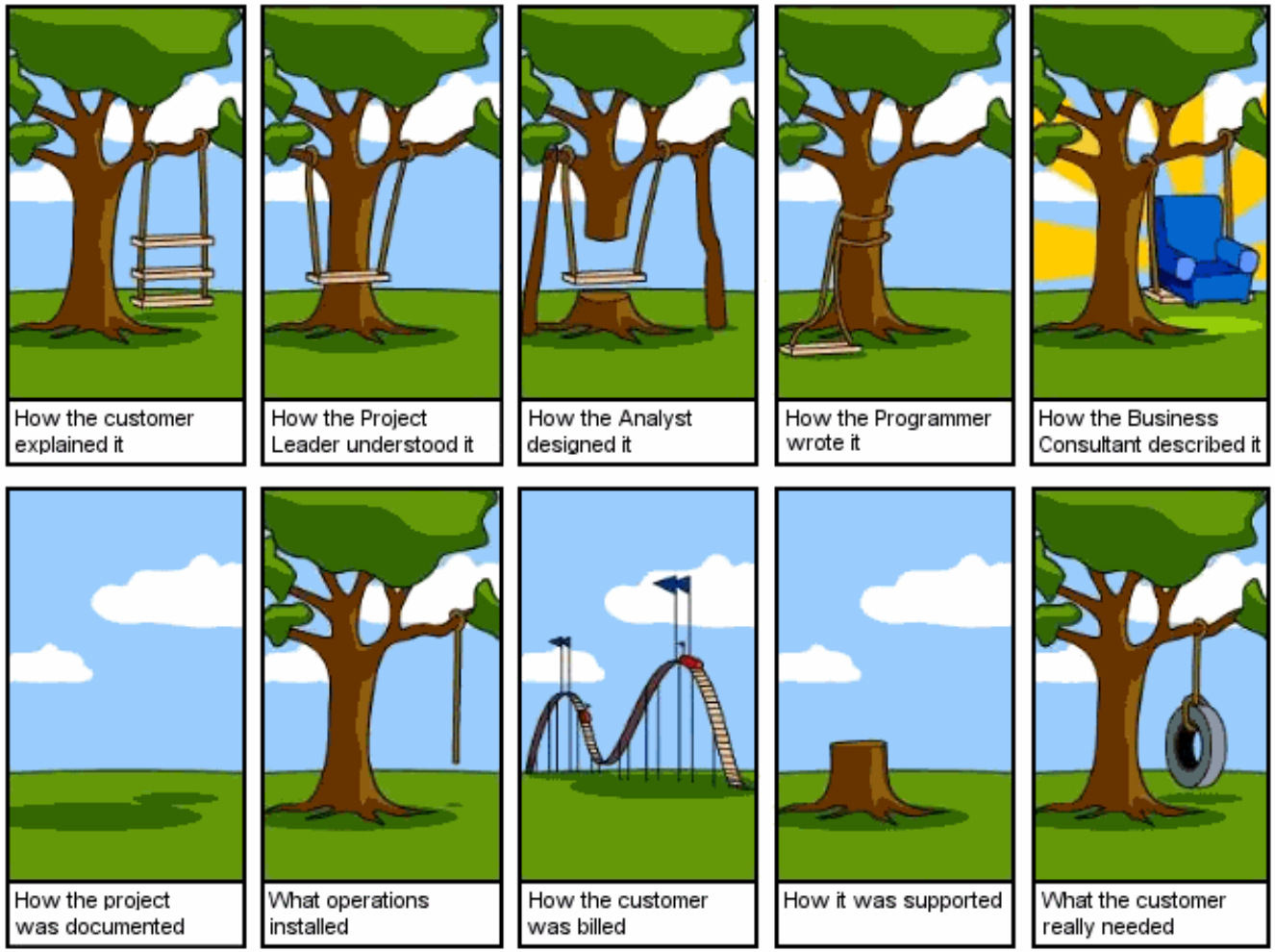
Requirements

zühlke
empowering ideas

- **You need them**
  - "The most critical risk facing most projects is the risk of developing the wrong product" – Mike Cohn
- **They will never be perfect**
  - Writing requirements that are unambiguous and complete is very hard or impossible (think of the Highway Code)
  - ….and understanding them can be even harder
- **They will always change (and change is good for all of us)**

Requirements

zühlke
empowering ideas



credited to Alex Gorbatchev on codinghorror.com

Non-functional Requirements

zühlke
empowering ideas

Often overlooked:

- Performance
- Scalability
- Availability
- Security
- Disaster Recovery
- Accessibility
- Monitoring
- Management
- Auditability
- …other Runtime aspects

- Flexibility
- Extensibility
- Maintainability
- Interoperability
- Legal
- Regulatory
- Internationalisation
- …other Non-runtime

It has to be fast!

We need it in French!

http://www.codingthearchitecture.com/

User Stories

zühlke
empowering ideas

- A way of capturing requirements through simple concrete examples
- User stories are non-technical in nature
  - describe something the system should do
  - not how it should do it
- Equally understandable by all stakeholders
  - BA, QA, Product Owner, Developer, User
- Used in testing, estimation, prioritisation, planning
  - In fact, everywhere conversations about project take place
- A *user story* captures the essence of a requirement by giving an example
- Can be great as backlog items in Scrum

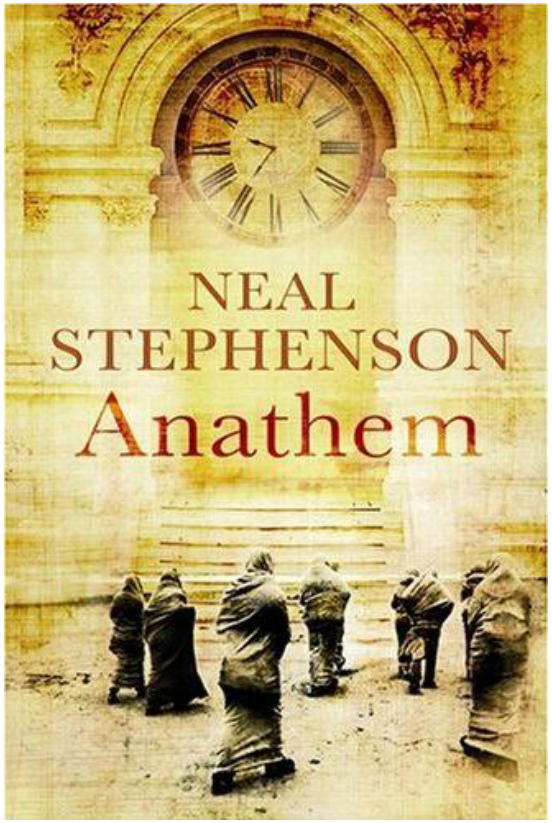
Time table
As a spectator
I want a daily time table of matches at Wimbledon
So that I can attend the matches that interest me

When a User Story is too big

zühlke
empowering ideas

When fleshing out a story, it may turn out to be at too high a level – it needs breaking down into shorter stories to fit into an iteration.

Such stories are known as *epics*.



How much design do you need?

The Design Spectrum

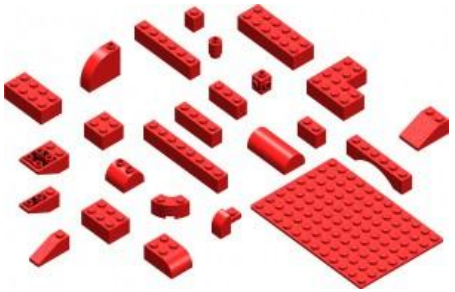
zühlke
empowering ideas

BDUF | Agile | Cowboy

- How detailed a design?
  - All classes and algorithms vs Key elements only
- Do complete design first?
  - Big Design Up Front vs Incremental vs Evolutionary
- How to capture the design?
  - Full UML model vs self documenting code
- Model Driven Development (MDD)

Design Essentials

- **Technology decisions**
  - Language, frameworks/libraries, deployment
- **Decomposition to modules/components**
  - High cohesion; modules have clear and focussed responsibilities
  - Good Abstraction; implementation details hidden
  - Enables use in a variety of configurations
- **Define common cross-cutting concerns**
  - Do not want multiple different mechanisms for logging, error handling, audit, security, persistence, configuration, initialisation etc.



Who is the design for?

zühlke
empowering ideas

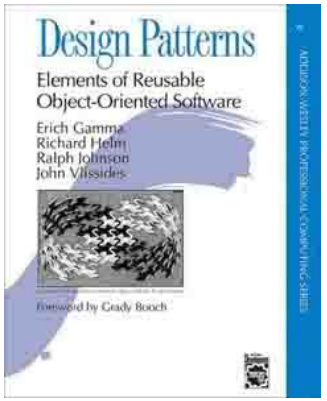
Agreement between a small team of developers?	A high level decomposition showing modules and key interactions may be sufficient; use simple UML and a whiteboard
Stakeholder approval?	A functional design clearly explaining responsibilities of each module and how non–functionals are addressed is recommended
Maintenance developers?	Will look at the code rather than detailed documentation. Would benefit from an overview, extension examples and having attention drawn to key areas
Deployment teams?	Description of any configurability, via run-time parameters, configuration files, or user interface is essential. Dependencies, logging and debug mechanisms also important
Agreement between teams on a large multi-partner project?	Better fully document all code interfaces on the boundary between partners and put under version control. Full API and behaviour needed. Painful

Design Patterns

zühlke
empowering ideas

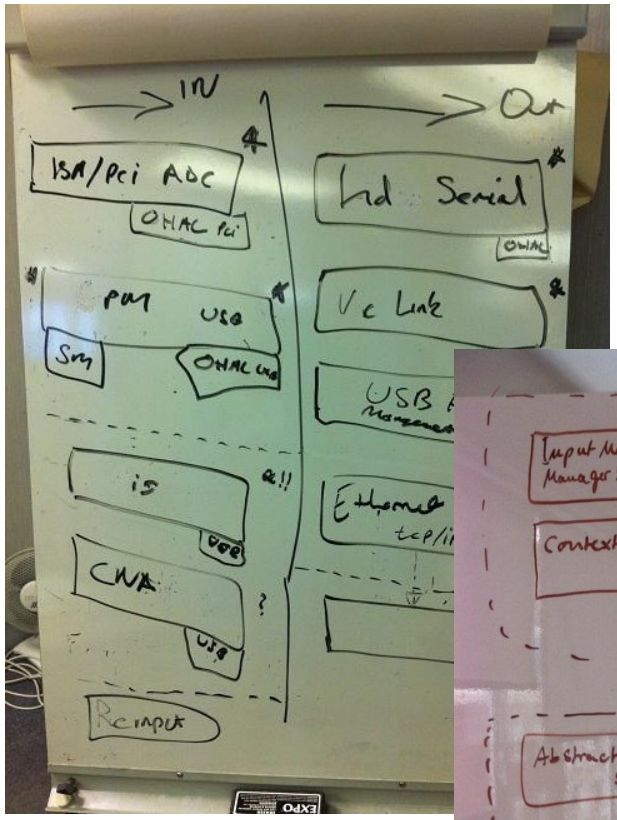
- **Reusable solution to a commonly occurring problem**
- **Useful for defining a common vocabulary**
- **Some patterns we seem to use a lot:**

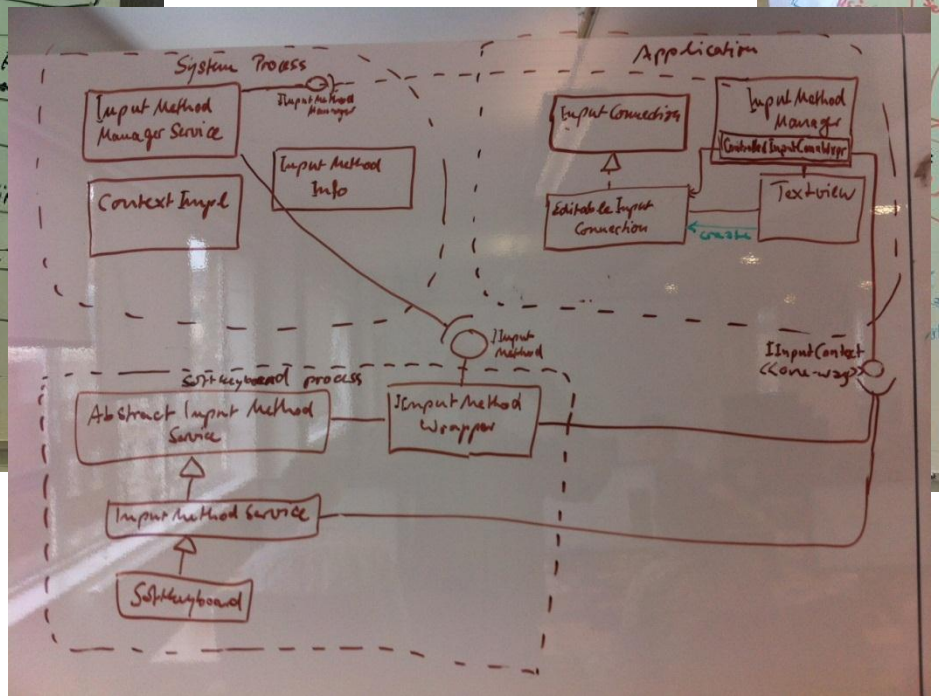
Observer	Event notification mechanism; observers register with a publisher who calls them back (notifies) on a specific state change
Iterator	Access a container's objects sequentially. Java and C++ collections provide iterators
State & Strategy	Change the behaviour of an object while maintaining the same interface
Factory patterns	Create an instance of an object letting the factory decide the appropriate concrete type

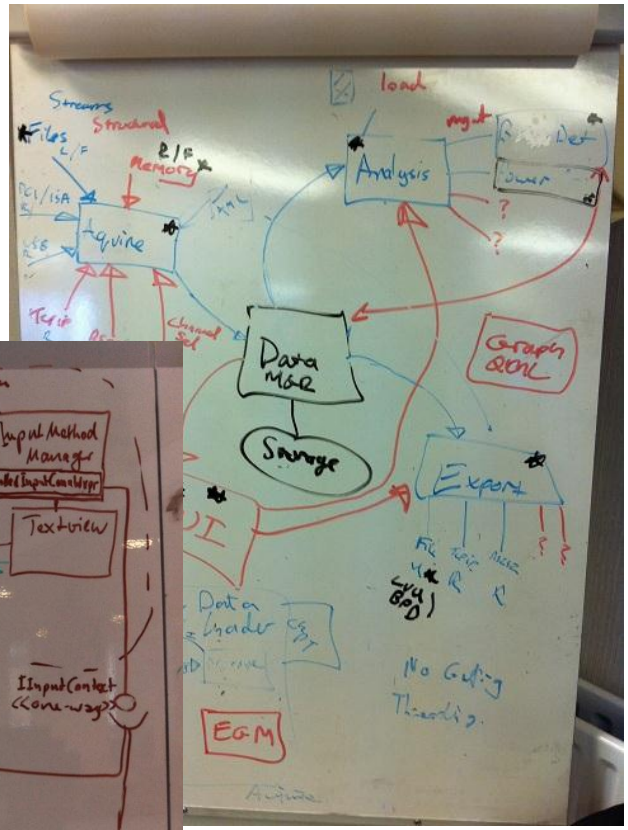


Design Capture Example – Whiteboards

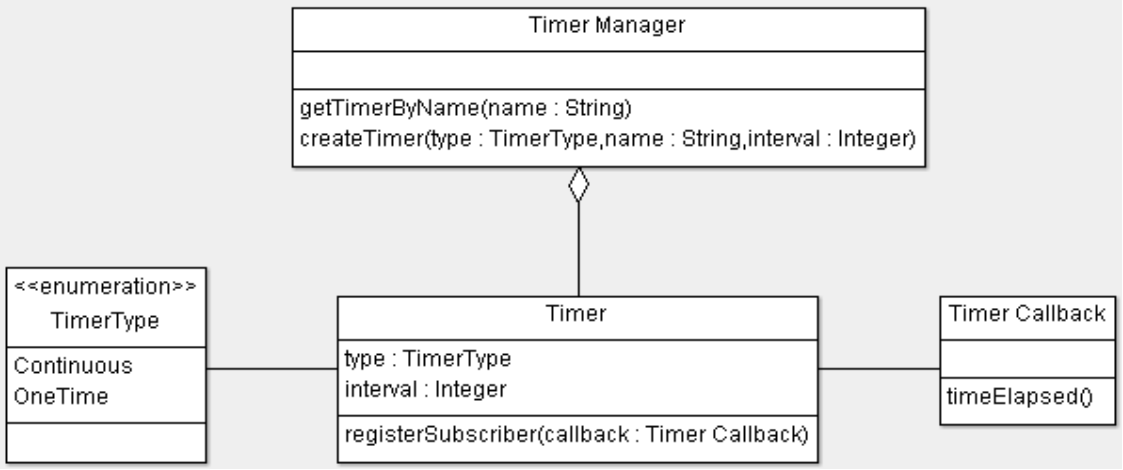
zühlke
empowering ideas

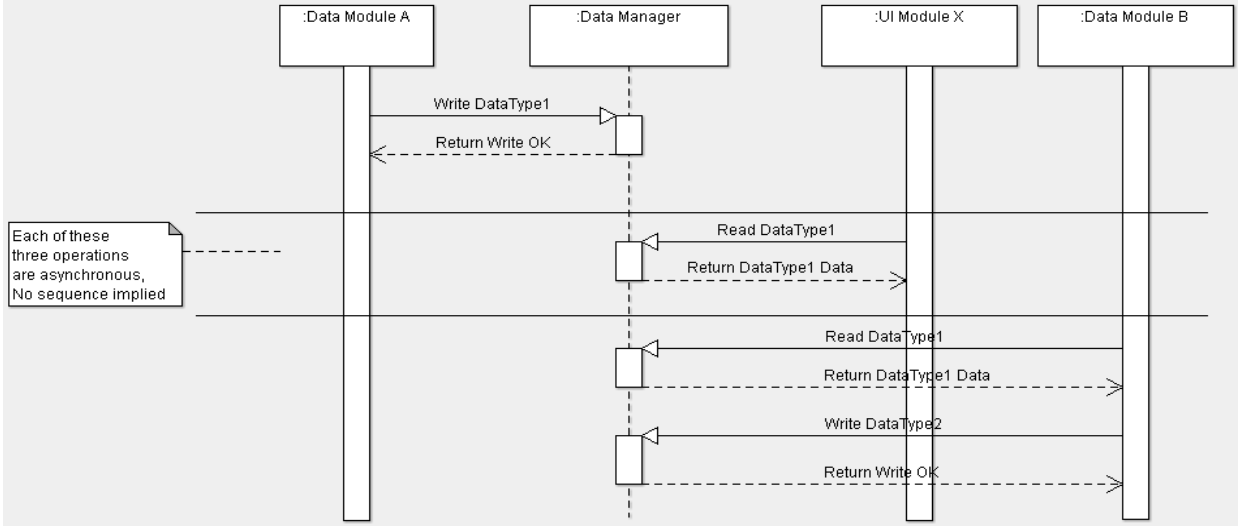






Design Capture Example – Simple UML





Design Capture Example - Text

zühlke
empowering ideas

6.2. Configurable Behaviour

The Serial Port supports the following configurable parameters:

Table 4 – Serial Port Configuration

Parameter Name	Description	Default Value
SerialPortNum	The number of the UART serial port to connect to	N/A
SerialPortBR	Serial Port Baud Rate to use	N/A
SerialPortDB	Serial Port Data Bits to use	8
SerialPortPAR	Serial Port Parity to use. Valid values are: • 'n' – none • 'e' – even • 'o' – odd • 'm' – mark • 's' – space	'n' – none
SerialPortSB	Serial Port Stop Bits to use	1
SerialNumber	Serial number to be transmitted in protocol messages	"0000-00-0000"

4.3.5. Timer Manager

The Timer Manager provides delayed call-backs to modules that need polled or time delayed behaviour. Some timed call-backs are totally independent, for instance polling of a hardware interface. Some timed call-backs need to be synchronised with others, for instance refresh of data on the user interface, so that a consistent data set is presented. The Timer Manager provides "named" timers for the latter type of clients.

zühlke
empowering ideas

Low Level Design

Mike Hogg

Low Level Design

- **Is more often than not code**
  - Exception may be extremely complex or time critical algorithms
- **Self-documenting code uses human-readable names for classes, methods, variables, etc**
  - Avoid abbreviations and generic names
  - IDE name completion means no penalty in long names

```
int process(int a[], int len) {
  int sum = 0;
  for(int i=0; i < len; i++) sum += a[i];
    sum = sum/len;
  return sum;
}
```

```
int calculateAverage(int values[], int arrayLength) {
  int sum = 0;
  for(int index=0; index < arrayLength; index++) {
    sum += values[index];
  }
  int average = sum/arrayLength;
  return average;
}
```

Documentation Generators

zühlke
empowering ideas

- Generate documentation from code annotations
  - Examples: Javadoc, Doxygen
- Can be useful - lower risk of going stale, but easily abused

```
/*! \brief Starts the timer.
 * \param sec timer seconds
 * \param msec timer milliseconds
 * \return void
 */
void StartTimer(unsigned long sec, unsigned long msec);
```

```
/*! \brief Starts the Hardware Timer. The abstract method TimerTick
 *  will be called every sec * 1000 + msec milliseconds until the
 *  StopTimer method is called
 * \param sec timer period, seconds component
 * \param msec timer period, milliseconds component
 * \return void
 */
void StartTimer(unsigned long sec, unsigned long msec);
```

SOLID OO Design Principles

zühlke
empowering ideas

S	Single Responsibility	An object should have a single responsibility. A responsibility can be viewed as a reason to change; a class should have one, and only one reason to change.
O	Open/closed	Objects should be open for extension but closed for modification. Commonly met by using abstract base classes which retain interface but allow extension of functionality
L	Liskov Substitution	An object instance should be replaceable with a subtype instance without altering program correctness. For instance a Square deriving from a Rectangle may violate LSP
I	Interface segregation	Many client specific interfaces are better than one general purpose one. No client should be forced to depend on methods that it does not use
D	Dependency inversion	Depend upon abstractions, do not depend upon concrete objects. High level components should not depend on low level components; wire their dependencies at runtime

http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

zühlke
empowering ideas

Other stuff

Mike Hogg

Important Tools

- **An IDE**
  - Look for name completion, refactoring support, etc
  - We use Eclipse
- **Version Control system**
  - Daily check-ins are standard practice
  - We use Subversion
- **Continuous Integration Server**
  - Builds and runs tests for all dependent modules on check-in
  - We use Jenkins
- **Backlog visible to all**
  - Various task tracking / collaboration tools available
  - We've written our own, or we use Excel
- **Unit test framework**
  - We use Junit and cxxtest amongst others

Keep Code Tidy

zühlke
empowering ideas

- Team all use same (simple) coding standards
  - Code structure – braces, tabbing etc
  - Naming conventions
  - File and directory organisation
  - Principles of note, etc
- Purge cruft and commented out code
- Don't write code you don't need just yet
- Think twice about making code common until you need it in two places
- Fix all code warnings
- Avoid TODOs where at all possible. Fix it now
- All code should have unit tests

Elegance in Simplicity

zühlke
empowering ideas

- **Writing horribly complex code is easy**
  - Nobody should **ever** be measured by lines of code written!
- **Writing simple, easy to follow code is hard**
  - Simple as in economic, elegant and easy to follow
  - Simple as in singularity of purpose
  - Simple as in obviousness of behaviour
  - Not simplistic, as in lacking in functionality
- **It is worth the investment**

Thank you for listening

Mike Hogg
Embedded and Mobile Systems
Business Unit Lead

Zuhlke Engineering Ltd
43 Whitfield Street
London W1T 4HD
United Kingdom

Phone: 0870 777 2337