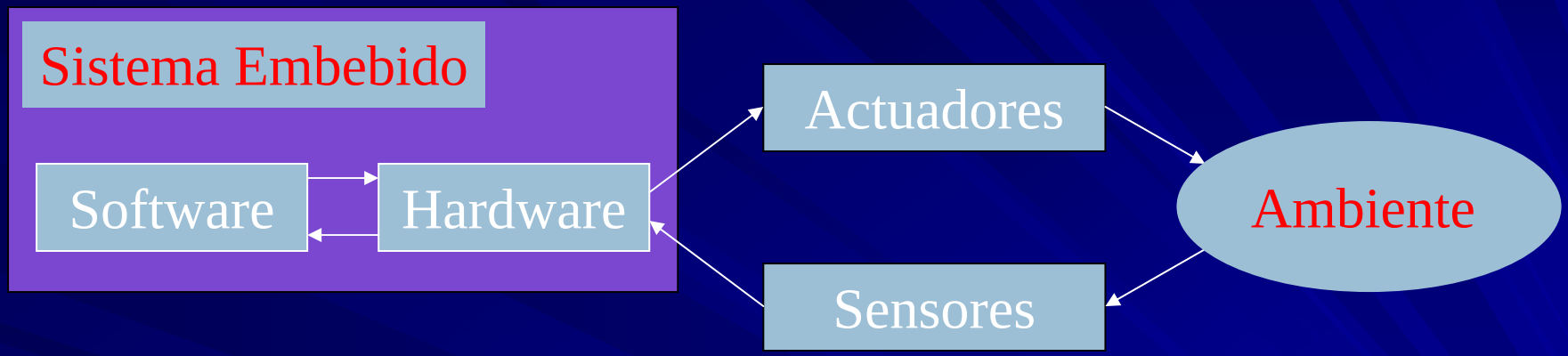
Programação de Sistemas Embebidos

Módulo 1
Introdução aos sistemas embebidos

João M. Fernandes

U. Minho & U. Algarve - Portugal

1. Características dos Sistemas Embedidos
2. Sistemas Tempo-Real
3. Orientação aos Objectos
4. UML
5. Motivação



- Um sistema embebido é qualquer dispositivo que inclua um computador ou processador programável, mas que não tenha por finalidade a computação genérica.

- 4 exemplos de sistemas embebidos

An example of:	Signal Processing	Mission Critical	Distributed	Small
Computing speed	1 GFLOPS	10 - 100 MIPS	1-10 MIPS	100,000 IPS
I/O Transfer Rates	1 Gb/sec	10 Mb/sec	100 Kb/sec	1 Kb/sec
Memory Size	32 - 128 MB	16 - 32 MB	1 - 16 MB	1 KB
Units Sold	10 - 500	100 - 1000	100 - 10,000	1,000,000+
Development Cost	$20M - $100M	$10M - $50M	$1M - $10M	$100K - $1M
Lifetime	15 - 30 years	20 - 30 years	25 - 50 years	10 - 15 years
Environment	Vibration, Heat	Heat, Vibration, Lightning	Dirt, Fire	Over-voltage, Heat, Vibration
Cost Sensitivity	$1000	$100	$10	$0.05
Other Constraints	Size, weight, power	Size, weight	Size	Size, weight, power
Safety	—	Redundancy	Mechanical Safety	—
Maintenance	Frequent repairs	Aggressive fault detection/maintenance	Scheduled maintenance	"Never" breaks
Digital content	Digital except for signal I/O	~½ Digital	~½ Digital	Single digital chip; rest is analog/power
Certification authorities	Customer	Federal Government	Development team	Customer; Federal Government
Repair time goal	1-12 hours	30 minutes	4 min. - 12 hours	1-4 hours
Initial cycle time	3-5 years	4-10 years	2-4 years	0.1-4 years
Product variants	1-5	5-20	10-10,000	3-10
Engineering allocation method	Per-product budget	Per-product budget	Allocation from large pool	Demand-driven daily from small pool
Other possible examples in this category:	Radar/Sonar Video Medical imaging	Jet engines Manned spacecraft Nuclear power	High-rise elevators Trains/trams/subways Air conditioning	Automotive auxilliaries Consumer electronics "Smart" I/O

Propriedades relevantes dum sistema embebido

- É desenvolvido para realizar uma função específica para uma dada aplicação (muitas vezes, será apenas posto a funcionar um único sistema).
- Deve apresentar um funcionamento em contínuo.
- Deve manter uma interacção permanente com o respectivo ambiente. Deve responder continuamente a diferentes eventos provenientes do exterior e cuja ordem e tempo de ocorrência não são previsíveis.
- Tem de ser correctamente especificado (e implementado), pois realiza tarefas críticas em termos de fiabilidade e segurança.
- Deve obedecer a imposições ou restrições temporais, pelo que questões de tempo-real têm que ser equacionadas.
- É digital.

Desenvolvimentos tecnológicos

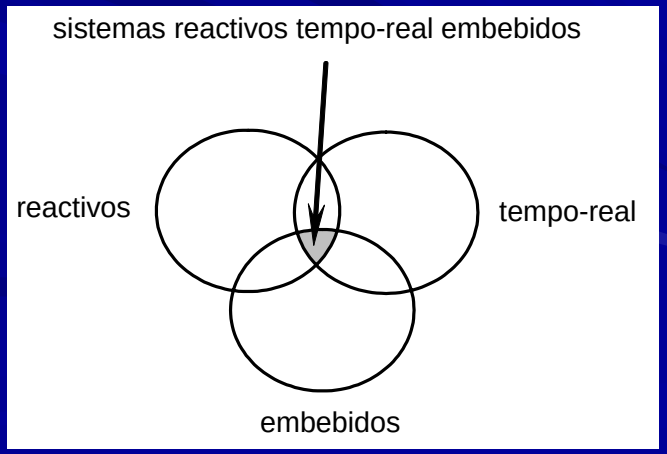
	Hardware	Software	Metodologias
1960	Computad. centrais Redes em estrela	S/ sistemas operativos Linguagem *assembly*	Procedimentos *ad-hoc*
		Execução multi-tarefa	Programação estruturada
1970	Mini-computadores Sistemas multi-comp. Barramentos	Linguagens de alto-nível Interfaces amigáveis Pacotes aplicacionais	Concepção estruturada Ferramentas CASE
1980	Micro-computadores Redes distribuídas	Sist. oper. distribuídos	Orientação por Objectos
		Linguagens tempo-real	Abordagem *life-cycle*
1990	Arq. Reconfiguráveis		Co-projecto HW/SW

- Muitos acham que a maioria dos computadores está nas secretárias e corre Windows/Linux.
- Em termos de números, os sistemas embebidos existem em maior quantidade.
- Numa casa, podemos encontrar 1 ou 2 PCs.
- Mas, nessa casa, existem várias dezenas de dispositivos que integram um processador.
- Na indústria, a presença das aplicações embebidas é ainda mais forte.
- O software convencional não inclui tantos requisitos não-funcionais como o software embebido.
- Questões temporais, de desempenho, de segurança, de fiabilidade, de peso, de tamanho têm de ser tidas em consideração em aplicações embebidas.

- O software embebido tem que operar, muitas vezes, de forma contínua e em ambientes hostis.
- O funcionamento do software embebido tem de ser autónomo e rigoroso em termos temporais.
- Alguns sistemas embebidos tem o potencial de provocar grandes danos, se mal usados/programados.
- Muitos sistemas embebidos são sistemas de tempo-real.
- "*Hard deadlines*" são requisitos de desempenho que têm obrigatoriamente de ser cumpridos.
- Uma *deadline* falhada constitui um erro; dados atrasados são dados errados!
- Sistemas de tempo-real *soft* são afectados por restrições temporais médias.

- Exemplos de sistemas de tempo-real *soft*: bases de dados on-line e sistemas de reservas de voo.
- Nestes sistemas, dados atrasados são dados bons!
- Um sistema embebido contém um computador como parte dum sistema maior e não existe para fornecer computação genérica a um utilizador.
- Um sistema embebido implementa uma função específica.
- Os sistemas embebidos interagem directamente com dispositivos eléctricos e indirectamente com mecânicos.
- A programação dos sistemas embebidos requer manipulações de baixo nível.

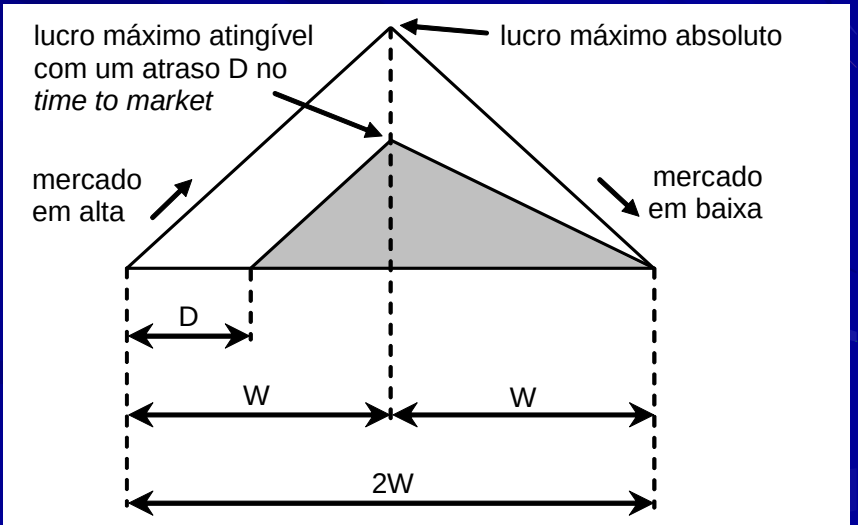
- O software convencional reage quase exclusivamente ao utilizador humano.
- Os sistemas embebidos têm de reagir ao utilizador, mas tem igualmente de interagir com sensores e actuadores.
- Um problema desta interacção com o ambiente é que o universo tem um comportamento imprevisível.
- Esses ambientes podem até ser adversos (indústria, meios aquáticos, medicina, ambientes de guerra)
- Os sistemas tem de reagir aos eventos quando eles ocorrem e não quando dá jeito!
- Muitos sistemas embebidos são, por natureza, reactivos.
- A resposta a eventos externos tem que ser totalmente fixa (*hard deadlines*).



- Muitos sistemas embebidos têm que executar uma (ou poucas) tarefas de alto nível.
- Para executar essas tarefas de alto nível, são precisas várias actividades de baixo nível, em simultâneo.
- Esta realidade implica concorrência.
- Sistemas uni-processador só executam uma instrução de cada vez, pelo que têm de implementar estratégias de escalonamento.
- Os sistemas embebidos têm de ser construídos com o processador mais barato que garante os requisitos de funcionalidade e de desempenho.
- Usar processadores com poucas capacidades baixa o custo, mas dificulta o desenvolvimento.
- Em software convencional não há problemas em criar um array com 1.000.000 de reais em vírgula flutuante.
- Podemos fazer o mesmo em aplicações embebidas?

- O desenvolvimento de aplicações embebidas é feito com ferramentas instaladas num PC, mas cujo alvo são plataformas mais pequenas e com menos capacidades.
- Esta realidade obriga ao uso de compiladores cruzados, que têm mais problemas que os compiladores nativos.
- Os recursos de hardware das plataformas (timers, conv. A/D, sensores) nem sempre são simuláveis num PC.
- Por vezes, também não existem ferramentas sofisticadas de teste/debug.
- Algumas plataformas nem um display disponibilizam para visualizar mensagens de erro.
- Não é raro ter que desenvolver software para correr em hardware que ainda não existe.

- Muitas aplicações embebidas têm que executar em contínuo, durante longos períodos de tempo.
- Nessas aplicações taxas de falhas graves idênticas ao Windows são totalmente inaceitáveis.
- Muitas aplicações são ainda críticas, tanto em termos de negócio, como de segurança.
- Dispositivos médicos, militares, industriais ou aviónicos têm de ser especialmente bem concebidos, pois uma falha tem efeitos catastróficos.
- Finalmente, o time-to-market dos sist. embebidos é baixo, o que impõe mais pressão nas equipas de desenvolvimento.



- O software para aplicações embebidas é mais difícil de desenvolver que aquele para aplicações de secretária.
- O software embebido tem todos os problemas do software convencional, mais muitos outros.

Questões para auto-avaliação (10 min.)

- Indicar 5 características dos sistemas embebidos.
- Dar exemplos de sistemas reactivos; sistemas tempo-real; sistemas embebidos; software convencional.
- Dar exemplos de sistemas embebidos nas 4 classes: processamento sinal; críticos; distribuídos e pequenos.
- Classificar o tipo de sistema de:
  - um “router” de rede
  - um telemóvel
  - um radar
  - um controlador dum conjunto de elevadores dum hotel
  - um gravador de vídeo (VCR)
  - uma consola de jogos (tipo PlayStation)
  - um PC (c/ impressora) para uso numa oficina de automóveis

- Um aspecto crítico de muitos sistemas embebidos é a forma como o tempo é tratado.
- Um sistema tempo-real é aquele que satisfaz restrições dos seus tempos de resposta a eventos externos.
- O desenvolvimento dum sistema tempo-real tem de identificar os requisitos temporais e garantir que o sistema está correcto funcional e temporalmente.
- Existem 3 tipos de sistemas tempo-real:
  - Forte (*hard*): o não cumprimento dum tempo de resposta a um evento conduz a uma falha do sistema.
  - Frouxo (*firm*): tolerada uma baixa probabilidade no não cumprimento dum tempo de resposta que conduz a uma falha.
  - Fraco (*soft*): o não cumprimento dum tempo de resposta a um evento conduz a uma degradação do desempenho.

- Muitos sistemas tempo-real usam RTOS.
- As funções dum RTOS são equivalentes às dum SO:
  - Gerir a interface com o hardware;
  - Escalonar e tarefas;
  - Gerir a memória;
  - Disponibilizar serviços comuns, incluindo I/O para dispositivos.
- Os RTOS divergem dos SO nos seguintes aspectos:
  - Escalabilidade;
  - Estratégias de escalonamento;
  - Suporte para ambientes embebidos e *diskless*.
- Os RTOS são escaláveis, o que significa que são estruturados por camadas.
- A camada mais interior (*kernel*) providencia as funcionalidades essenciais do RTOS.

- Funcionalidades adicionais podem ser incluídas, à medida que forem necessárias.
- A escalabilidade torna um RTOS usável tanto em aplicações para plataformas pequenas, como em soluções grandes e distribuídas.
- Esta arquitectura denomina-se microkernel, para dar ênfase ao tamanho minimalista do kernel.
- Os RTOS tipicamente oferecem políticas de escalonamento com preempção baseada em prioridades.
- Neste tipo de escalonamento, as tarefas com maior prioridade “ultrapassam” as tarefas com menor prioridade, quando aquelas estão disponíveis para correr.

- Em sistemas tempo-real, o desempenho médio é um aspecto secundário.
- O principal é que o sistema cumpra todos os requisitos temporais, mesmo no pior cenário possível.
- RTOS foram criados para soluções embebidas.
- Habitualmente, é possível fazer o boot a partir duma ROM, o que se revela útil para sistemas sem disco.

- As aplicações embebidas são muito diversas em tamanho e âmbito.
- Aplicar uma metodologia genérica torna-se muito difícil.
- As metodologias OO não são “varinhas mágicas”, mas apresentam diversas vantagens:
  - Consistência das vistas e dos respectivos modelos;
  - Melhoria das abstracções do domínio do problema;
  - Maior estabilidade na presença de mudanças dos requisitos;
  - Facilidade de reutilização;
  - Facilidade de escalabilidade;
  - Suporte para questões de fiabilidade e segurança;
  - Suporte directo a concorrência;
- Com objectos, não há uma melhoria dramática, mas é possível construir sistemas com um melhor relaciona-mento ao mundo real.

Consistência das vistas e dos respectivos modelos

- Um dos problemas das abordagens estruturadas é a difícil transição entre a análise e a concepção.
- Não é trivial mostrar a relação entre as vistas de análise (DFDs e ERDs) e as de concepção (structure charts).
- Mais complicado ainda é indicar qual o fluxo de dados ou o processo implementado por um pedaço de código.
- Em OO, o mesmo conjunto de vistas de modelação é usado em todas as fases de desenvolvimento.
- Objectos e classes identificados na fase de análise têm representação directa no código.
- Com objectos, A relação entre o problema e a solução é mais evidente.

Melhoria das abstracções do domínio do problema

- Outro dos problemas das abordagens estruturadas é a limitada abstracção e baixo encapsulamento.
- Há uma forte separação entre estrutura e comportamento o que limita fortemente a utilidade dos modelos.
- Os modelos dos objectos mantêm uma forte coesão entre os dados e as operações que os manipulam.
- As abstracções baseadas em objectos são mais intuitivas e tem maior capacidade de modelação.
- O vocabulário para designar os objectos provêm do domínio do problema (e não da solução).
- Utilizadores e pessoas do marketing podem melhor perceber as implicações dos seus requisitos, pois os modelos usam os seus conceitos e a sua linguagem.

Maior estabilidade na presença de mudanças

- Em software, uma pequena mudança nos requisitos pode ter efeitos catastróficos na estrutura do software.
- Abstracções baseadas em separação entre dados e processos mostram-se muito instáveis.
- As mudanças nos programas são evitadas, tornando os produtos mais rapidamente obsoletos.
- Os sistemas OO, por basearem as suas abstracções no mundo real, tendem a ser mais estáveis.
- Em princípio, a estrutura fundamental dos problemas não muda muito rapidamente.
- Alterações aos requisitos implicam adicionar ou remover aspectos ao modelo, mas não a sua reestruturação.

Facilidade de reutilização

- A reutilização em programação estruturada é conseguida modificando directamente o código fonte.
- O paradigma OO inclui 2 mecanismos para facilitar a reutilização: a generalização e o refinamento.
- A generalização (i.e. herança) permite adicionar e estender as características dum componente, sem fazer qualquer alteração ao seu código.
- O refinamento possibilita a especificação de compo-nentes, que são depois refinados através da adição das partes em falta.
- Por exemplo, pode definir-se um algoritmo de ordenação e depois refiná-lo ara diferentes tipos de dados.

Facilidade de escalabilidade

- O ponto chave de qualquer método para desenvolver software é o controlo da complexidade.
- Em OO, o elevado nível de abstracção e encapsula-mento, implica uma baixa dependência entre objectos.
- O uso dos mesmos conceitos (os objectos), ao longo do desenvolvimento, facilita o progresso dos projectos.

Suporte para questões de fiabilidade e segurança

- Devido ao elevado nível de encapsulamento, em OO a relação entre componentes é limitada e bem-definida.
- Este facto aumenta a fiabilidade, pois o controlo da interacção entre objectos é mais simples.
- É possível ainda explicitar pré e pós-condições que asseguram um melhor funcionamento do sistema.
- Por exemplo, em C é o cliente dum array quem tem de garantir que não se usa um índice fora dos limites.
- Em Java, esse teste é feito pelo próprio objecto array.
- Os sistemas OO disponibilizam ainda tratamento de excepções, para assegurar que condições excepcionais são tratadas correctamente.

Suporte directo a concorrência

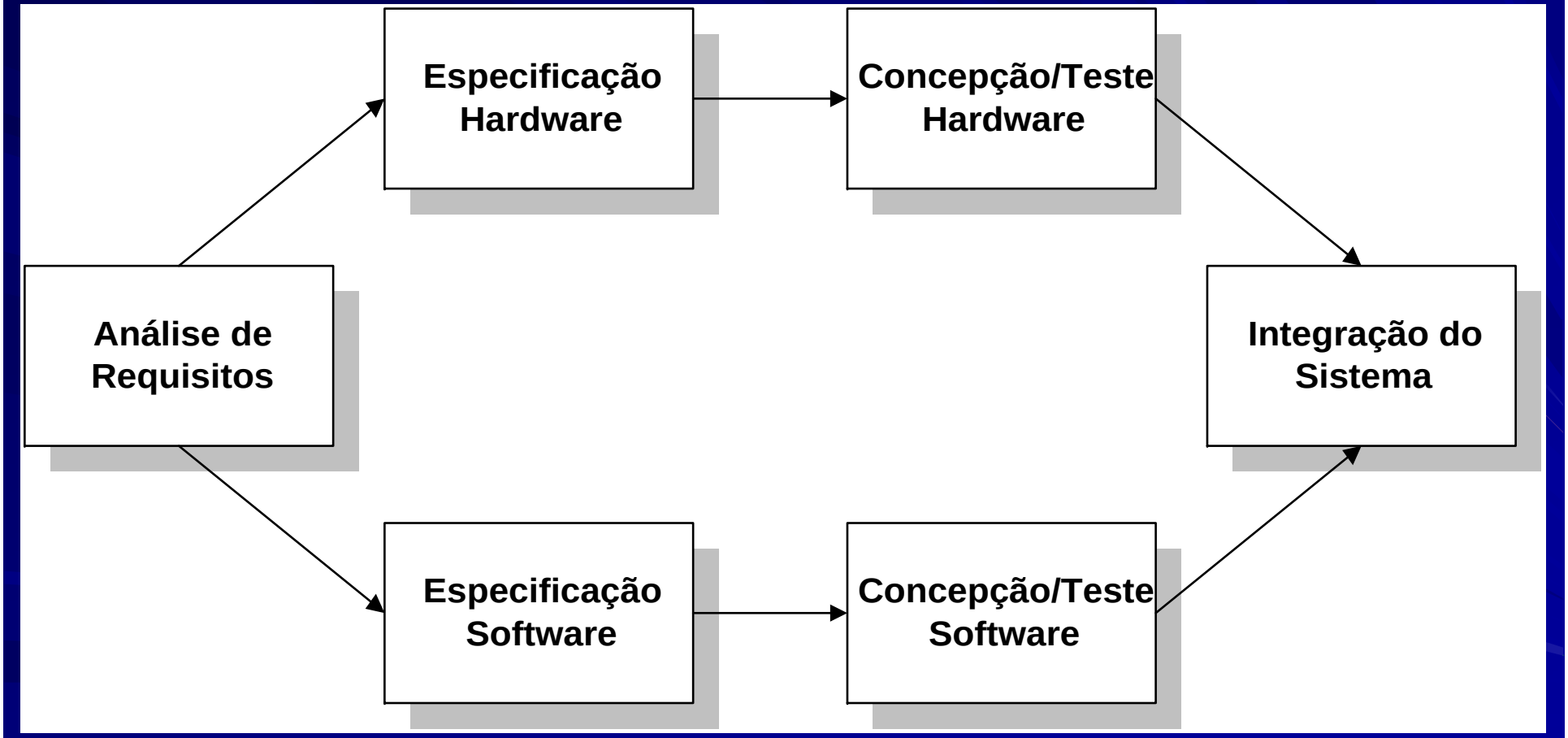
- A concorrência é uma característica muito importante dos sistemas embebidos.
- Nos métodos estruturados, não há noção de concorrência, nem de gestão e sincronização de tarefas.
- Os sistemas OO são por natureza concorrentes.
- As questões de sincronização podem ser explicitamente representadas por statecharts, objectos activos e mensagens entre objectos.

- As aplicações embebidas são muito diversas em tamanho e âmbito.
- Aplicar uma metodologia genérica torna-se muito difícil.
- As metodologias OO não são “varinhas mágicas”, mas apresentam diversas vantagens.
- Um metodologia é composta por 3 componentes:
  1. Processo
  2. Métodos
  3. Notação (ou linguagem)
- Actualmente a notação UML é um standard de facto e é usada em diversos áreas tecnológicas.
- A sua aplicação aos sistemas embebidos é recente, mas tem vindo a ganhar popularidade.

- UML é apenas uma notação. **NÃO** é:
  1. um processo
  2. um método
  3. uma metodologia
- A notação UML é multi-vista, pois é composta por vários diagramas.

- Nesta disciplina, vamos aprender a usar UML para desenvolver sistemas embebidos.
- Mostra-se ainda que métodos e processos devem ser seguidos para tirar vantagens de UML.

O modelo de processo tradicional



- Algumas crenças
  - A interligação é fácil.
  - O software é maleável (i.e. fácil de alterar), pelo que deficiências encontradas no hardware podem rectificar-se no software.
  - O desenvolvimento das 2 componentes faz-se em paralelo.
- Na prática
  - Abordagem baseada no software.
  - Abordagem baseada no hardware (a mais usual)

- Típica abordagem baseada no hardware
  - Escolha dum microprocessador (ou microcontrolador) comercial. Esta opção é ditada pelo conhecimento específico dos projectistas no referido processador.
  - Desenvolvimento duma versão *bread boarded* do hardware a projectar, sob o qual o software irá ser desenvolvido.
  - Projecto do hardware e produção de software (decorrem paralela e independentemente). Surge uma primeira versão do hardware final.
  - Integração do software e do hardware desenvolvidos. Esta fase produz muitos problemas e nela são detectadas inúmeras falhas, de que resultam alterações profundas, no hardware e no software.
  - Finalmente, após várias iterações e reformulações, é possível construir um protótipo do sistema em desenvolvimento.

- Problemas da abordagem baseada no hardware
  - O tempo de desenvolvimento é muito prolongado.
  - O produto final não responde aos requisitos do utilizador. (análise do problema muito superficial).
  - A documentação dos diversos elementos do projecto é precária.
  - A codificação é pouco metódica (código esparguete).
  - A integração do software no hardware é muito “dolorosa”.
  - A detecção de erros não é facilitada.
  - É impossível alterar o hardware (passar funcionalidades do software para o hardware, depois da placa estar feita).
  - A abordagem não produz um sistema optimizado (quaisquer que sejam as métricas usadas).
  - A manutenção é muito difícil.

- O que é preciso para desenvolver sistemas embebidos
  - Modelos de especificação multi-vista (UML).
  - Processo de software controlável e repetitível.
  - Métodos de projecto.
  - Abordagem orientada aos objectos.
  - Mecanismos para atacar a complexidade dos sistemas.

- Nesta disciplina, apresentam-se as necessidades metodológicas para desenvolver sistemas embebidos de grande complexidade, pondo especial ênfase nas questões associadas à fase de análise e à modelação.