

OU3

RadioInfo

Emmy Lindgren
emli0277@ad.umu.se
id19eln@cs.umu.se

4 januari 2023

Innehåll

1	Introduktion	1
1.1	Krav på uppgiften	1
2	Användarhandledning	1
2.1	Hur startar man programmet?	1
2.2	Gränssnitt	2
2.3	Hur använder man programmet?	4
3	Sveriges radio API	4
4	Systembeskrivning	5
4.1	Arkitektur	5
4.2	Systemets uppbyggnad	5
4.2.1	Programmets uppstart	5
4.2.2	Ladda in och visa kanaler	6
4.2.3	Ladda in och visa tablå	6
4.2.4	Visa mer information om ett program	8
4.3	Trådsäkerhet	8
4.4	Designmönster inom systemet	9
4.4.1	Observer/Observable	9
4.4.2	Adapter	9
4.4.3	Template method	9
5	Begränsningar	10
6	Vidareutveckling	10
7	Reflektion	11

1 Introduktion

För denna uppgift i kursen Applikationsutveckling (Java) ska ett program implementeras för att presentera Sveriges radios kanaler och dess respektive tablåer. För att genomföra detta ska Sveriges Radios API användas för att hämta data om kanalerna, dess tablåer och programmen som tablån innehåller.

Syftet med uppgiften är att få insikt i hur XML/JSON kan implementeras och användas, samt träna på att skapa egna GUI:n, skriva trådsäkra program, följa ett givet dataformat och öva färdighet i rapportskrivning.

1.1 Krav på uppgiften

Tablåinformation ska laddas ned första gången som en användare vill visa kanalens tablå, och sedan uppdateras en gång i timmen. Det ska också finnas möjlighet för användare att manuellt välja att uppdatera informationen. Om en användare väljer att visa tablåinformation från en kanal som redan laddat ned tablåinformation ska ingen ny information laddas ned, utan den cachade informationen visas istället. Vid nedladdning av tablåinformation ska alla program 6 timmar före och 12 timmar efter tidpunkten för nedladdning läggas in i tablån.

I tablån ska finnas information om programmen visas, dess namn, start- och sluttid. Användare ska också på något sätt kunna visa mer information om ett specifikt program, inkluderande beskrivning och bild.

Utöver detta ska all uppdatering av data från API:et ske i bakgrunden utan att det grafiska gränssnittet blockeras, och sedan uppdatera det grafiska gränssnittet med den inlästa datan. Programmet ska också använda sig av en menybar.

2 Användarhandledning

2.1 Hur startar man programmet?

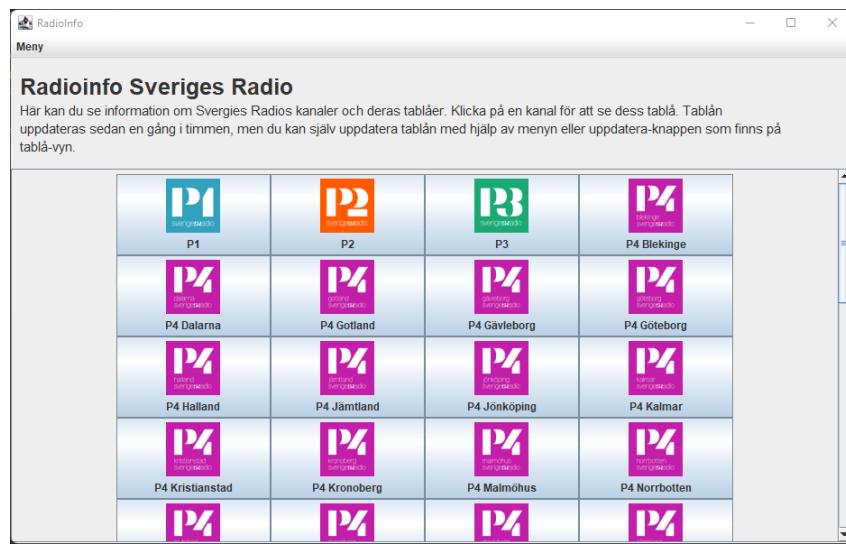
För att skapa en jar-fil av programmet används IntelliJ. I menyn File -> Project Structure -> Artifacts kan man klicka på en plus-ikon. Sedan Add -> JAR -> From modules with dependencies. Där får man välja main-klassen för programmet, sedan klicka ok. Nu är en jar-fil definierad och den behöver nu byggas. Välj Build -> Build Artifacts -> main-klassnamn.jar -> Build. Då byggs jar-filen. Denna behöver byggas om varje gång något är ändrat i koden i programmet.

När en jar-fil finns kan programmet köras. I kodpaketet som laddats upp tillsammans med denna rapport finns en färdig jar-fil under Out -> Artifacts. Starta programmet genom att skriva in kommandot `java -jar RadioInfo.jar` i en kommandotolk.

2.2 Gränssnitt

Programmet består av två huvudvyer, en kanalvy för att visa de kanaler som går att välja mellan och en tablåvy för att visa tablån för vald kanal. I båda vyer finns en menyrad med ett val, meny. Där finns menyalternativen *Alla kanaler*, *Uppdatera sidan* samt *Avsluta*.

I kanalvyn visas information om programmet i en slags header, och efter detta kommer alla kanaler som finns tillgängliga i ett rutnät av knappar, en för varje kanal. Se figur 1.



Figur 1: Kanalvy

I tablåvyn visas information om den valda kanalen i en slags header, och därefter visas kanalens tablå i en tabell. Här färgas cellerna för start- och sluttid olika beroende på om de är avslutade, pågår eller kommande vid tidpunkten för hämtningen av informationen. Ovan tablån finns två knappar, en informations-knapp och en knapp för att uppdatera sidan. Se figur 2.

Program	Starttid	Sluttid
P3 Nyheter	tors, 29 dec. 11:00	tors, 29 dec. 11:02
P3 med Tina Mehratsoon	tors, 29 dec. 11:02	tors, 29 dec. 12:00
P3 Nyheter	tors, 29 dec. 12:00	tors, 29 dec. 12:02
P3 med Tina Mehratsoon och Edvin Edvinsson	tors, 29 dec. 12:02	tors, 29 dec. 13:00
P3 Nyheter	tors, 29 dec. 13:00	tors, 29 dec. 13:02
P3 med Edvin Edvinsson	tors, 29 dec. 13:02	tors, 29 dec. 14:00
P3 Nyheter	tors, 29 dec. 14:00	tors, 29 dec. 14:02
P3 med Edvin Edvinsson	tors, 29 dec. 14:02	tors, 29 dec. 15:00
P3 Nyheter	tors, 29 dec. 15:00	tors, 29 dec. 15:02
P3 med Edvin Edvinsson	tors, 29 dec. 15:02	tors, 29 dec. 16:00

Figur 2: Tablåvy

För att visa mer information om ett program i en tablå används en dialogruta som öppnas ovanpå huvudvyn. Det ser ut enligt figur 3.

Program	Sluttid
Morgonpasset i P3 med David Druid, Kodjo Akolor och Margret Atladottir	tors, 29 dec. 08:32
	tors, 29 dec. 09:00
	tors, 29 dec. 09:02
	tors, 29 dec. 10:00
	tors, 29 dec. 10:02
	tors, 29 dec. 11:00
	tors, 29 dec. 11:02
	tors, 29 dec. 12:00
	tors, 29 dec. 12:02
	tors, 29 dec. 13:00

Figur 3: Programinformations-vy

Om något går fel under programmets gång och användaren behöver bli informerad används även då en informationsruta. Den ser ut enligt figur 4, med varierande textinnehåll.



Figur 4: Informationsruta

2.3 Hur använder man programmet?

När programmet startat kommer man först till kanalvyn. Här kan man i menyraden välja att avsluta programmet, *Alla kanaler*-alternativet tar användaren till samma vy som denne är på nu, och *Uppdatera sidan*-alternativet är avaktiverat för denna vy för att visa att denna sida inte går att uppdatera. I själva kanalvyn kan man välja en kanal för att visa dennes tablå. Väljer en kanal gör man genom att klicka på den. Om kanalen inte har någon associerad tablå informeras användaren genom informationsrutan.

Tablåvyn visar information om kanalen samt dess tablå. I menyraden kan man nu välja att uppdatera sidan, eller gå tillbaka till kanalsidan genom att välja *Alla kanaler*. Även här fungerar menyalternativet *Avsluta*. Ovan tablån finns två knappar. Knappen med i:et visar information som kan vara bra att veta om tablåvyn, så som vilka program som laddas in och hur man kan se mer om dem. Knappen med ladda-ikonen uppdaterar sidan på samma sätt som *Uppdatera sidan*-menyvalet i menyn. Informationen på sidan laddas om en gång i timmen. Om något går fel i inläsningen kommer den att sluta uppdatera en gång i timmen, tills dess att användaren väljer att gå in på den kanalens tablåvy igen.

Programmens namn, start- och sluttid visas för användaren. Start-och sluttiden visas i grått om programmet redan gått, rött om det går just nu och grönt om det ännu inte har gått. För att se mer information om ett program kan man dubbelklicka på den raden i tabellen. Den informationen stänger man med hjälp av kryssset längst upp till höger i informations-rutan.

3 Sveriges radio API

För uppgiften hämtades all information från Sveriges Radios API (Sveriges Radio, 2021). Metoderna kanaler samt tablå användes för denna uppgift. Metoden Kanaler kan användas för att hämta information om kanaler. Där får man kanalens id, dess namn, färg, tagline, en länk till liveljud från kanalen, en länk till kanalens bild, samt en länk till kanalens tablåinformation. Kanalens typ får man också, riks-, extra- eller lokalkanal.

Metoden Tablå hämtar kanal- samt programtblåer. För varje program i tablån får man programmets id, titel, start- och sluttid, url till episoden, programmet samt kanalen som programmet hör till samt länk till programmets bild. I API-förfrågan anger man vilken

kanal och vilket datum man vill hämta tablån för. Om inget datum anges hämtas tablån för dagens datum.

Attribut i båda metoder kan vara felaktiga eller inte heller finnas med i svaret, vilket måste hanteras. Sidohantering är också på vid en vanlig förfrågan, vilket betyder att man måste i så fall bläddra sig genom alla sidor i svaret för att få hela svaret. Detta kan stängas av med hjälp av attributet `pagination=false`.

4 Systembeskrivning

4.1 Arkitektur

Vid implementation har MVC använts, Model-View-Controller. Arkitekturen syftar till att separera modell och vy-objekt för att skapa en så modulär kod som möjligt (JavaT-Point, 2021). I mitten finns en controller, som sköter kommunikationen mellan vy och modell. Vid en ren MVC-implementation ska modell och vy inte ha någon vetskapp om varandra.

Modellen representerar objekten som bär data och innehåller logik relaterad till den datan. Vyn är det som presenteras i applikationen, som i detta fall det grafiska gränssnittet som visas upp med kanaler och tablåer. Här ska modellen visualiseras. Controller ligger mellan dessa, och sköter själva flödet i applikationen, uppdaterar modell och vy med data till och från varandra.

4.2 Systemets uppbyggnad

Systemet är implementerat med Java 17. Det startas på EDT, Event Dispatch Thread. Med Java Swing bör allting som har med Swing att göra köras på EDT på grund av att Swing inte är trådsäkert (Oracle, 2022). Skulle man hantera Swing-klasser på andra trådar än EDT kan man skapa trådsäkerhetsproblem. Så det första som sker i programmet, i mainklassen, är att en huvudcontroller-instans startas på EDT. För att göra detta används koden:

```
SwingUtilities.invokeLater(Controller::new);
```

Det är sedan controllern som skapar en instans av vyn, som i sin tur bygger det grafiska gränssnittet som visas för användaren. På så sätt vet inte vyn om controllern, och det är controllern som äger vyn. Controllern sätter också de lyssnare som behövs i vyn med hjälp av publika metoder som finns i vy-klassen.

4.2.1 Programmets uppstart

Programmet startas som sagt direkt på EDT med huvudcontrollern **Controller**. Denne skapar då en instans av huvudvyn, **MainView**. Vyn bygger då de två olika vyerna och placeras dem i en panel med en *CardLayout*. De placeras då i panelen tillsammans med en nyckel i form av en sträng. På så sätt kan man med nyckelns hjälp enkelt byta mellan de

två olika vyerna när det behövs. Det görs enligt `cardLayout.show(panel, "nyckel");`. Controllern sätter sedan lyssnare på vyn, i menyn, på kanalerna samt på tabellen som kommer att visa tablån.

4.2.2 Ladda in och visa kanaler

För att representera kanaler finns modellen **Channel**. Modellen innehåller all information om en kanal som behövs i programmet; namn, tagline, tablåURL, bild samt själva tablån också. Modellen håller också koll på om det finns någon cachad information i tablån eller inte.

För att visa kanalerna i vyn används en `JList`. Men för att kunna visa en lista av modellerna (kanalerna) till vyn behövs en annan modell, `AbstractListModel`, en slags adapter mellan modell och vy (läs mer under 4.4.2). En egen modell som extendar `AbstractListModel`, vilken innehåller kanaler skapades därför, **ChannelListModel**. Den håller en lista av kanalobjekt, som man kan addera till. Det finns också funktioner definierade för att hämta listans längd och en kanal på en specifik rad i listan. `JList`-objektet tar alltså in en `ChannelListModel` för att veta vad listan ska rita upp. Inbyggt i `JList` finns också lyssnare som sätts på `ChannelListModel`, enligt observermönstret (läs mer under 4.4.1). På så sätt uppdateras listan i vyn om något läggs till i modellen. Här skapas en koppling mellan vy och modell utan att skapa beroenden.

I systemet skapas denna `ChannelListModel` redan i controllern och skickas med till vyn för att användas vid skapandet av `JList` som håller alla kanaler. Den skickas sedan vidare till en Swingworker som vars jobb är att ladda in kanalerna på just den modellen, **ChannelWorker**. `ChannelWorker` skickar en API-förfrågan till SRs API om dess kanaler. När svaret kommer så skickas varje elementnod med information om en enskild kanal in till konstruktorn i `Channel`, som skapar ett `Channel`-objekt från det. Konstruktorn i `Channel` laddar även in kanalens bild. När ett `Channel`-objekt är skapat används metoden `publish` för att lägga till kanalen i `ChannelListModel`. Då uppdateras även vyn direkt med den nya kanalen.

För att kunna visa innehållet i ett `channel`-objekt i vyn så använder sig `JList` även av en `ListCellRenderer`. En egen implementation av denne, **ChannelRenderer** skapades också för att visa kanalerna som knappar. **ChannelRenderer** plockar ut kanalernas namn och bild och visar sedan dem i form av en knapp med lite extra design. Denna sätts på `JList` genom `list.setCellRenderer(new ChannelRenderer());`.

4.2.3 Ladda in och visa tablå

För att representera ett program som finns i en tablå finns modellen **Program**. Modellen håller all information om ett program som behövs i programmet, dess namn, start- och sluttid, beskrivning, bild och status. Status representerar om programmet har gått, går eller ska gå vid tidpunkten för hämtning.

För att visa tablån i vyn används en `JTable`. Även här behövs en adapter mellan modell

och vy, en AbstractTableModel. Därför skapades **ProgramTableModel** som extender AbstractTableModel. ProgramTableModel samlar en lista med program, alltså en tablå, för att sedan visas i en JTable, som en tabell. Metoder som `getValueAt(row, column)` och `getColumnName(column)` måste implementeras för att JTable ska veta hur den ska rita upp tabellen i vyn. Även JTable sätter lyssnare på AbstractTableModel, och uppdaterar i vyn om något lagts till i modellen. När tablån för en kanal ska uppdateras behöver systemet bara uppdatera listan som finns i ProgramTableModel (tablån) och kalla på `fireTableDataChanged()` så meddelas lyssnarna, alltså vyn, och denne kan uppdatera sig med det nya datat. ProgramTableModel är implementerad så att den tar in en lista av program och ersätter sin interna lista med denna. Datat byts alltså helt ut.

I systemet så får varje kanal en egen ProgramTableModel som symbolisera deras tablå redan i dess konstruktör. Den är tom till en början, men ska hålla det cachade datat. Då en användare klickar på en kanal, så plockar kontrollern ut den valda kanalen ur JList:en, och hämtar först tablåURL:en ur denna. Om ingen URL finns så meddelas användaren om att det inte finns någon tablå för vald kanal. Annars hämtas tablån (ProgramTableModel) ur kanalen. Kontrollern ser sedan till att vyn ändras till tablå-vy och information om den valda kanalen sätts i headern för den vyn, samt att modellen för JTable sätts till tablån som just plockats ut ur kanalen. Eftersom att alla kanaler vid konstruktion får en tom ProgramTableModel kan vyn ändras och sättas innan en tablå ens har lästs in. När tablån sedan har lästs in så kan programmet uppdatera modellen och på så sätt även vyn.

Efter att vyn har uppdaterats så kontrolleras om kanalen har en cachad tablå eller inte. Har kanalen ingen cachad tablå så startas en ScheduledExecutorService för att skapa och köra en SwingWorker, **TableauWorker**, först en gång direkt och efter det en gång i timmen. TableauWorker tar in kanalen, vyn och även ScheduledExecutorService-instansen som skapats. Den sistnämnda tas in för att kunna avsluta de automatiska uppdateringarna en gång i timmen ifall något går fel vid inläsning. TableauWorker använder sig sedan av modellklassen **ApiTableauParser** för att hämta en lista med program.

ApiTableauParser tar in en URL från vilken denna ska läsa in program på. Först kontrolleras tiden på dygnet just nu, om klockan är före 6 på morgonen så vet vi med säkerhet att vi måste hämta gårdagens samt dagens tablå. På liknande sätt vet vi att om klockan är efter 12 på dagen så måste vi hämta dagens samt morgondagens tablå. Annars räcker det med att hämta dagens tablå. När tablån för de olika datumen hämtas så skapas ett program-objekt för varje program genom att skicka varje elementnod till konstruktorn för Program. Sedan kontrolleras ifall programmet ligger inom 6 timmar före eller 12 timmar efter hämtning, först då läggs det till i listan av program. Denna lista returneras sedan till TableauWorker.

TableauWorker hämtar listan med program och uppdaterar kanalens tablå med den listan, och vyn uppdateras med det hämtade datat. Sedan meddelas kanalen även att denne nu har cachad data, så ingen mer ScheduledExecutorService behöver startas. Detta görs i done, så alltså på EDT. Om något går fel vid inläsningen så informeras användaren om

det härifrån också.

Kontrollern sätter också nya lyssnare på knapparna som ska uppdatera sidan, så att dessa vid en knapptryckning uppdaterar rätt tablå. Då startas en instans av TableauWorker för detta också, men här skickas null in istället för ScheduledExecutorService, eftersom att det nu inte finns något att stänga av om något går fel.

4.2.4 Visa mer information om ett program

För att visa mer information om ett specifikt program behöver användaren dubbelklicka på det valda programmet i tabellen. Vid registrering av ett dubbelklick hämtar kontrollern modellen i tabellen och ur denna hämtas det valda programmet. Sedan kontrolleras om programmet har en cachad bild eller inte. Om ingen cachad bild finns så startas en SwingWorker, **ProgramImageWorker** för att hämta bilden från API:et. ProgramImageWorker tar in ett CallBackInterface. Det är ett functional interface där det definieras en metod, `callback()`, som ska kallas då swingworkern är klar. Från kontrollern skickas då en runnable in som det argumentet, vilket kommer att bli vad swingworkern egentligen kollar på i done.

Där skickas in ett metodkall till metoden `showProgramInfo(program)` som finns i kontrollern. Denna metod säger till vyn att skapa en dialogruta med information om programmet. Så en swingworker startas för att ladda in bilden, och sedan uppdatera vyn med programmets information. Det går väldigt fort så man märker det knappt som användare, men kontrollern sätter också en väntar-muspekare medan swingworkern jobbar. Detta skapar en bättre användarupplevelse. Är det så att programmet redan har en inladdad bild cachad så kallas metoden på direkt istället och ingen swingworker behöver startas.

4.3 Trådsäkerhet

När man jobbar med Swing så måste man tänka på att Swing inte är trådsäkert. Allting som uppdaterar vyn måste därför göras på EDT. Huvudkontrollern i programmet startas på EDT av just den anledningen, att mycket av det som görs i kontrollern är att uppdatera och förändra vyn och måste göras på EDT. När kanalerna ska laddas in görs detta på en Swingworker för att inte låsa vyn, men kanalerna läggs till i ChannelListModel allt eftersom att dem laddas in med hjälp av Publish och Process i swingworkern, som körs på EDT.

När det kommer till att ladda in tablåerna så ska detta göras mycket oftare än kanalerna. Men samma princip gäller även här. En swingworker används, men denna gången för att samla en hel lista med program. När swingworkern är klar så körs done-metoden i swingworkern på EDT, och det är först där som ProgramTableModel uppdateras med listan av program som laddats in. Det görs alltså på EDT, vilket betyder att flera trådar inte kan uppdatera ProgramTableModel samtidigt. Eftersom att hela listan sätts in och byts ut, så kan inte program från olika inläsningar visas samtidigt i vyn heller.

Vid inläsning av bilder till kanalerna så görs detta vid inläsningen av kanalerna, alltså på swingworkerens bakgrundsträd. Bilder till programmen läses in med en separat swingworker, som sedan i done (på EDT) kollar på en metod som uppdaterar vyn med programdata. Här får användaren vänta en liten stund för att se datat men vyn fryser inte, utan inläsning sker ändå i bakgrunden.

4.4 Designmönster inom systemet

4.4.1 Observer/Observable

Designmönstret Observer är ett mönster för beteende (behavioral). Med hjälp av mönstret kan man skapa prenumerationer på ett visst objekt. De som är intresserade av att veta när en förändring skett hos objektet kan starta en prenumeration. Det innebär att prenumeranter får en notis när någonting sker, utan att den som dem prenumererar på behöver veta vilka det är som prenumererar (Refactoring Guru, 2022a). Ett enkelt sätt att visa att något har skett, utan att skapa starka beroenden mellan objekt.

Observer-mönstret används inom systemet i JList och JTable. Dessa lyssnar på deras respektive modeller, ChannelListModel samt ProgramTableModel. I modellerna används metoderna `fireIntervalAdded` och `fireTableDataChanged` för att notifera alla prenumeranter om att någon förändring skett i modellen. När JList och JTable får den notisen så läser de av datat igen och uppdaterar det som har förändrats i vyn.

4.4.2 Adapter

Designmönstret adapter är ett mönster för struktur (structural). Adapter är ett sätt att få objekt med olika interfaces att samarbeta (**refactoringGuruAdapter**). Det gör adapter genom att översätta mellan objekten, så att objekt 1 kan få data på den form som den kräver, medan objekt 2 kan skicka data på det sätt som den kan.

I systemet används adapter-mönstret för att översätta mellan modell och vy, i form av ChannelListModel samt ProgramTableModel. Dessa tar in Channel- respektive Program-objekt och sparar dem. När JList respektive JTableModel sedan vill visa upp datat som finns i kanaler och tablåer, så finns i modellerna implementerade metoder som svarar mot de metoder som JTable och JList kräver för att kunna rita upp en lista och en tabell. Metoderna använder sig av Channel- och Program-objekten för att hämta det som ska returneras.

4.4.3 Template method

Designmönstret template method är ett beteendemönster (behavioral). Med hjälp av template method kan man definiera en mall av algoritmer i en superklass, som man sedan kan skriva över i subklasser, utan att förändra strukturen (Refactoring Guru, 2022b). På så sätt kan man bryta ned en algoritm i mindre delar, varav delarna i sig kan implementeras på olika sätt men ändå hålla algoritmen intakt.

I systemet används template method-mönstret mycket. Exempelvis i huvudvy-klassen. Där är konstruktorn väldigt liten, bestående av nästan enbart metod-kall till metoder som bygger upp vyn. Dessa går bra att ersätta med en annan metod som bygger den delen av vyn. Klassen **ImageLoader** är ett annat exempel, den används för att ladda in bilder. Den används på flera ställen i koden och går bra att ersätta med någon annan metod som läser in bilder från en URL.

5 Begränsningar

Den första begränsningen är att kanalerna laddar in ganska så långsamt som det ser ut just nu. De uppdateras eftersom och ger ändå användaren känslan av att det laddar dock. Det blev heller inte särskilt snyggt med knapparna för kanalerna, jag ville gärna ha lite mellanrum mellan dem. Det visade sig dock att det var allt annat än lätt när man jobbade med en JList och enbart fejkade knappar istället för celler i en lista.

Programmet uppdaterar inte heller status på programmen allt eftersom tiden går, denna läses in enbart då programmen hämtas. Det betyder att om en användare hämtar datat en viss tid så kommer ett program att markeras med rött eftersom att det går just vid hämtningen. I de flesta fall handlar det om ganska korta program, vilket betyder att det efter bara några minuter ofta inte är just det programmet som går något mer.

Till sist så meddelar programmet när det blir fel vid inläsningen av data. Dock är felmeddelandena ganska begränsade då det är svårt att veta vad som faktiskt har gått fel. Exempelvis om ett **IOException** kastas så kan detta betyda att användaren inte har någon internetuppkoppling och därfor inte kommer åt API:et. Men det kastas också om det blir något fel på serversidan eller om förfrågan till API:et är trasig. Exempelvis kastas ett sådant exception från kanalen SR Extra 15, då jag tror att den har en trasig länk eller bara en länk som är aktiv ibland.

6 Vidareutveckling

Som vidareutveckling ser jag några potentiella saker. Det första är att användare ska kunna uppdatera kanalvyn. Som det ser ut just kan användare inte göra det, vilket var ett designval av mig för att göra saker enklare. Kanaler håller den cachade datan om tablåer, vilket gör att om de laddas om så försvinner också det cachade datat om tablåer. Men detta kan nog lösas genom att uppdatera kanaler istället för att helt ladda om dem vid en eventuell omladdning. Det kan finnas tillfällen då en användare känner behov av att ladda om även kanal-sidan.

Det andra är att låta användare filtrera kanaler baserat på deras kanaltyp, det vill säga riks-, lokal- eller extrakanal. Man kanske inte vill se lokala kanaler från andra län än sitt eget, eller så vill man bara se rikskanalerna. Det tror jag skulle vara en stor förbättring och ge extra funktionalitet till programmet.

Till sist tror jag även att i tablåvyn skulle kunna snyggas till lite. När man först laddar

in tablån för en kanal så tror jag att man uppskattar om man får se det program som går just nu och de som går efter först, för att sedan kunna bläddra upp och se de program som redan gått. En automatisk scroll ned till det program som pågår just nu vore trevligt att ha. När extra information visas om ett program så tror jag även här att informationen kan utökas. Exempelvis skulle man kunna ha en länk till programmet så att användaren kan ta sig dit. Men också information om vilka kanaler som programmet går på, ibland går ett program på fler kanaler än en.

7 Reflektion

Denna uppgift har varit väldigt svår, men rolig. Jag tycker att det var bra att vi fick lära oss om hur man läser XML/JSON och speciellt om hur man kan koppla modeller till vyn utan att skapa beroenden. Det känns som att det finns väldigt många riktiga situationer som data behöver uppdateras ofta, och att det då inte är rimligt att be en kontroller att hela tiden uppdatera vyn varje gång som data ändras.

Det som har varit svårt, väldigt svårt faktiskt, med denna uppgift har varit trådarna. Det har bara varit en föreläsning dedikerad till trådar, vilket jag tycker är tråkigt. Det kändes som att jag skulle lära mig mer om trådar i denna kurs, men istället har jag fått försöka hitta information själv, vilket inte har varit det lättaste. Jag skulle ha uppskattat fler föreläsningstillfällen när det gäller trådar och Swing.

Referenser

- JavaTPoint. (2021). *MVC Architecture in Java*. <https://www.javatpoint.com/mvc-architecture-in-java> (hämtad: 29.12.2022)
- Oracle. (2022). *The Event Dispatch Thread*. <https://docs.oracle.com/javase/tutorial/uiswing/concurrency/dispatch.html> (hämtad: 29.12.2022)
- Refactoring Guru. (2022a). *Observer*. <https://refactoring.guru/design-patterns/observer> (accessed: 29.12.2022)
- Refactoring Guru. (2022b). *Template method*. <https://refactoring.guru/design-patterns/template-method> (accessed: 29.12.2022)
- Sveriges Radio. (2021). *Sveriges Radios öppna API - (version 2)*. <https://api.sr.se/api/documentation/v2/index.html> (hämtad: 29.12.2022)